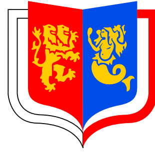

THE UNIVERSITY
OF BIRMINGHAM

Speech Recognition in Programmable Logic

by

Stephen Jonathan Melnikoff

A thesis submitted to

The University of Birmingham

for the degree of

Doctor of Philosophy

Electronic, Electrical and Computer Engineering

The University of Birmingham

November 2003

UNIVERSITY OF BIRMINGHAM

University of Birmingham Research Archive

e-theses repository

This unpublished thesis/dissertation is copyright of the author and/or third parties. The intellectual property rights of the author or third parties in respect of this work are as defined by The Copyright Designs and Patents Act 1988 or as modified by any successor legislation.

Any use made of information contained in this thesis/dissertation must be in accordance with that legislation and must be properly acknowledged. Further distribution or reproduction in any format is prohibited without the permission of the copyright holder.

The University of Birmingham

Electronic, Electrical and Computer Engineering

Degree of Doctor of Philosophy

in Electronic and Electrical Engineering

Speech Recognition in Programmable Logic

Stephen Jonathan Melnikoff BEng(Hons) MSc MIEE

PhD Thesis

Revision 75 — 27th October 2003

Final Edition (Electronic Version)

E-mail: s.j.melnikoff@iee.org

University No.: 311254

Supervisor: Dr Steven F Quigley

©2003 Stephen Jonathan Melnikoff

Abstract

Speech recognition is a computationally demanding task, especially the decoding part, which converts pre-processed speech data into words or sub-word units, and which incorporates Viterbi decoding and Gaussian distribution calculations.

In this thesis, this part of the recognition process is implemented in programmable logic, specifically, on a field-programmable gate array (FPGA).

Relevant background material about speech recognition is presented, along with a critical review of previous hardware implementations. Designs for a decoder suitable for implementation in hardware are then described. These include details of how multiple speech files can be processed in parallel, and an original implementation of an algorithm for summing Gaussian mixture components in the log domain. These designs are then implemented on an FPGA.

An assessment is made as to how appropriate it is to use hardware for speech recognition. It is concluded that while certain parts of the recognition algorithm are not well suited to this medium, much of it is, and so an efficient implementation is possible.

Also presented is an original analysis of the requirements of speech recognition for hardware and software, which relates the parameters that dictate the complexity of the system to processing speed and bandwidth.

The FPGA implementations are compared to equivalent software, written for that purpose. For a contemporary FPGA and processor, the FPGA outperforms the software by an order of magnitude.

For M&D

ls

PhD. . . Doctor of Philosophy, earned by several years' postgraduate study regardless of subject title, perhaps because at the end of ten years in higher education, you have to be philosophical about the value of it all.

– Robert Ainsley, *Bluff Your Way at University*

Acknowledgments

I would like to thank:

- My friends, for making these five years in Brum such an enjoyable experience;
- My parents, for their unwavering support;
- Jonathan Bilmen, BMedSci, PhD, for his friendship, ceaseless exploits, and for learning far more about speech recognition and programmable logic than any biochemist/medic ever should;
- And finally: Dr Steven Quigley, supervisor extraordinaire, for his assistance, advice, good humour in the face of adversity (e.g. undergraduates, impending deadlines, paperwork), and startlingly brightly coloured ("tasteful"[1]) shirts.

[1]Source: Dr S Quigley

Contents

1 Introduction **1**

1.1 Aims and objectives 2

1.2 Speech recognition 2

1.2.1 Overview 2

1.2.2 Speech pre-processing 3

1.2.3 HTK 5

1.3 Field-programmable gate arrays 6

1.4 Motivation 9

1.5 Contribution 11

1.6 Structure of the thesis 12

2 Speech recognition theory **14**

2.1 The speech recognition problem 14

2.2 Hidden Markov models 16

2.3 Viterbi decoding 18

2.3.1 Decoding 18

2.3.2 Termination & backtracking 19

2.3.3 Assumptions 20

2.3.4 Log-domain representation 21

2.4 Language model 22

2.5 Discrete HMMs . . . 23
2.6 Continuous HMMs . . . 24
2.7 Gaussian mixture summation (log-add algorithm) . . . 26
2.8 Duration modelling . . . 27
2.9 Summary . . . 28

3 Speech recognition in hardware **29**
3.1 Accuracy measures . . . 30
3.2 Parallel systems . . . 31
3.2.1 Custom ICs . . . 31
3.2.2 SIMD arrays . . . 32
3.2.3 MIMD arrays . . . 35
3.2.4 Associative string processors . . . 36
3.3 Serial systems . . . 37
3.4 FPGAs . . . 38
3.5 Commercial products . . . 40
3.6 Alternative recognition methods . . . 41
3.7 Gaussian mixture summation . . . 42
3.8 Other hardware implementations . . . 43
3.8.1 Convolutional decoding . . . 43
3.8.2 Traceback . . . 45
3.8.3 Training . . . 45
3.8.4 Trees . . . 46
3.9 Summary . . . 46

4 System design **47**
4.1 Structure . . . 48

4.2 Data representation 49
4.2.1 General 49
4.2.2 HTK 49
4.3 Viterbi decoder 50
4.3.1 Initialisation and switching 50
4.3.2 Scaler 51
4.3.3 Language model 51
4.3.4 HMM block 52
4.3.5 Padding buffer 54
4.4 Observation probability computation 54
4.4.1 Parallelism 54
4.4.2 Gaussian mixture components 55
4.4.3 Data storage 57
4.5 Gaussian mixture summation 58
4.5.1 Top-level structure 58
4.5.2 Data analysis & design 59
4.6 Duration modelling 61
4.6.1 Parallel architecture 61
4.6.2 Serial architecture 63
4.7 Full language model 63
4.8 Further design issues 65
4.8.1 Control 65
4.8.2 Pruning 66
4.8.3 Non-emitting states 67
4.9 Summary 67

5 Implementation **69**

5.1 System environment 69
5.1.1 Hardware 69
5.1.2 Software 70
5.1.3 Speech data & models 75
5.1.4 Virtex/Virtex-E FPGAs 77
5.2 Disc1: discrete, monophones 77
5.2.1 49-HMM implementation 78
5.2.2 7-HMM implementation 78
5.2.3 1-HMM implementation 79
5.3 Cont1: continuous, monophones 80
5.4 Cont1P: continuous, monophones, parallel 82
5.5 Cont3P: continuous, biphones/triphones, parallel 83
5.6 Cont3_4: continuous, biphones/triphones, parallel, Gaussian mixtures . . 83
5.7 Cont3_4D1: continuous, biphones/triphones, Gaussian mixtures, hardware/software hybrid 84
5.8 Cont3_4D: continuous, biphones/triphones, Gaussian mixtures, duration modelling, software 85
5.9 Summary 85

6 Requirements analysis 88
6.1 Number of HMMs 89
6.2 Number of HMMs implemented 89
6.3 Number of mixture components & number implemented 90
6.4 Language model 90
6.5 Duration modelling 91
6.6 Parallel files 92
6.7 Results 92

6.8 Summary 94

7 Results 96

7.1 Hardware vs software 96

7.1.1 Virtex/Virtex-E vs Pentium-III 450 (debug mode) 98

7.1.2 Virtex/Virtex-E vs Pentium-III 450 (release mode) 101

7.1.3 Virtex/Virtex-E vs Athlon XP 2000+ (release mode) 102

7.1.4 Virtex-II (projected) vs Athlon XP 2000+ (release mode) 104

7.2 Resource usage 105

7.3 Recognition rates 106

7.4 Summary 110

8 Conclusions 111

8.1 Key conclusions 112

8.1.1 Suitability 112

8.1.2 Hardware vs software 112

8.1.3 Requirements 113

8.2 Summary 114

8.3 Further work 114

8.4 The future. 116

References 118

Appendix: Publications 127

Figures

2.1 HMM finite state machine . . . 17
2.2 HMM trellis . . . 17

4.1 System structure . . . 48
4.2 Recogniser structure . . . 48
4.3 Viterbi decoder core . . . 50
4.4 Node structure . . . 53
4.5 Observation probability computation block . . . 55
4.6 Gaussian mixture component (GMC) block . . . 56
4.7 Gaussian mixture summation block . . . 58
4.8 Log-add table structure . . . 60
4.9 Log-add structure . . . 60
4.10 Proposed architecture for explicit duration modelling . . . 62
4.11 Proposed architecture for language model block . . . 64

5.1 Software input and outputs . . . 71
5.2 Software object model . . . 73

Tables

5.1 Speech models . . . 76
5.2 Summary of implementations . . . 87

6.1 Summary of effects on time per observation and bandwidth . . . 95
6.2 Summary of effects on speedup . . . 95

7.1 Timing: Virtex/Virtex-E vs Pentium-III 450 (debug mode) . . . 100
7.2 Timing: Virtex/Virtex-E vs Pentium-III 450 (release mode) . . . 100
7.3 Release dates for FPGAs and processors under test . . . 102
7.4 Timing: Virtex/Virtex-E vs Athlon XP 2000+ (release mode) . . . 103
7.5 Timing: Virtex-II (projected) vs Athlon XP 2000+ (release mode) . . . 103
7.6 Resource usage . . . 107
7.7 Recognition rates . . . 108

Acronyms & abbreviations

ASIC	application-specific integrated circuit
BRAM	block RAM
CLB	configurable logic block
CORDIC	co-ordinate rotation digital computer
DSP	digital signal processor/processing
DTW	dynamic time warping
FF	flip-flop
FIFO	first-in, first-out (queue)
FPGA	field-programmable gate array
FSM	finite state machine
GMC	Gaussian mixture component (block) [made up by the author]
HMM	hidden Markov model
HTK	HMM toolkit
IC	integrated circuit ("chip")
I/O	input/output
IP	intellectual property
LUT	look-up table
MFCC	mel-frequency cepstral coefficient
MIMD	multiple instruction, multiple data
PC	personal computer
PCB	printed circuit board
PCI	peripheral component interconnect
p.d.f.	probability density/distribution function
PE	processing element
PLD	programmable logic device
RAM	random-access memory
ROM	read-only memory
RTR	run-time reconfiguration
SIMD	single instruction, multiple data
SoC	system on a chip
SRAM	static RAM
SRL	shift register (shift right logical)
VHDL	very high speed integrated circuit hardware description language

Roman symbols

a_{ij}	transition probability: probability of a transition from state i to state j
$b_j(\mathbf{O}_t)$	observation probability: probability of state j emitting the observation $\mathbf{O}_t$ at time t
c_{jm}	mixture weight for state j and mixture component m
D	maximum duration
$d_j(\tau)$	duration probability: probability of staying in state j for duration τ
H	total number of HMMs
i	previous state
j	current state
K	constant multiplier for log-domain probabilities
L	number of elements in observation feature vector and Gaussian vectors
l	current element
M	number of mixture components in Gaussian mixture
m	current mixture component
$\mathcal{N}$	normal (Gaussian) distribution
N	total number of states in all HMMs
$\mathbf{O}$	observation sequence $\mathbf{O} = \mathbf{O}_0, \mathbf{O}_1 \ldots \mathbf{O}_{T-1}$
$\mathcal{P}$	probability
P^*	probability of most likely final state
p	word insertion penalty
Q	state sequence $q_0, q_1 \ldots q_{T-1}$
Q^*	most likely state sequence $q_0^*, q_1^* \ldots q_{T-1}^*$
s	grammar scale factor
T	length of sequence (maximum time)
t	current time

Greek symbols

$\delta_t(j)$	maximum probability, over all partial state sequences ending in state j at time t, that the HMM emits the sequence $\mathbf{O} = \mathbf{O}_0, \mathbf{O}_1 \ldots \mathbf{O}_t$
λ^*	the set of all models representing spoken utterances (word or sub-word unit)
λ	a model sequence, representing a sequence of spoken utterances
μ	mean
$\nu_t(i, j, \tau)$	probability at time t, of the system having moved to state j at time $t - \tau + 1$ from state i at time $t - \tau$, and then having stayed in state j for duration τ
$\xi_t(j, \tau)$	probability at time t, of the system having moved to state j at time $t - \tau + 1$ from its most likely predecessor state at time $t - \tau$, and emitting the observation sequence from time $t - \tau + 1$ to t
π_j	probability of being in state j at time $t = 0$
σ	standard deviation
σ^2	variance
τ	current duration, i.e. length of time spend in the current state
$\psi_t(j)$	most likely predecessor state of state j at time t

Speak properly, and in as few words as you can, but always plainly; for the end of speech is not ostentation, but to be understood.

William Penn (1644–1718)

1 Introduction

"Computer! Write me a PhD thesis!" The idea of being able to talk to a computer, and have it understand you, has been a recurring theme in science fiction for decades. While we are not yet at the stage where computers can comprehend our every word, and act on them, these machines are becoming ever more complex and ubiquitous.

But before a computer (or, for that matter, a human being) can attempt to understand speech, it must first convert the audio stream it receives into what that stream actually represents: initially, the basic sounds that make up a language, and ultimately, words. To do that with greater reliability and fidelity, and to be able to cope with different speakers and noisy environments, are the goals of current research in speech recognition.

While others concentrate on developing the algorithms and models, there still remains the question of how to implement them. Commercial software packages already exist which can run on a PC — but they are limited by having to operate on a general-purpose processor. In the end, to achieve the maximum processing power, application-specific hardware is the answer. Accordingly, in this thesis, a hardware implementation of a speech recognition system is presented.

1.1 Aims and objectives

The aim of this research is to design and implement a speech recognition system, with the decoding stage implemented in hardware, in order to:

- assess the suitability of so doing for the various parts of the recognition algorithm;
- compare the processing speed of hardware and software implementations, in order to ascertain the possible speedup;
- determine the requirements inherent in applying hardware to speech recognition.

The hardware in question is a field-programmable gate array (FPGA).

1.2 Speech recognition

1.2.1 Overview

A typical speech recognition system consists of three stages. The first is the pre-processing stage, described in more detail below, which takes a speech waveform as its input, and extracts from it feature vectors or observations which represent the information required to perform recognition.

The second stage is recognition, or decoding, which is performed using a set of statistical models called hidden Markov models (HMMs). At their simplest, the HMMs represent monophones, i.e. the basic distinct sounds of a particular language, of which English has around 50. However, when people speak, these sounds are affected by those uttered immediately before and after them. In order to model this effect, a larger number of models, now representing pairs and triplets of monophones (biphones and triphones), can be used, leading to improved recognition ability. In addition, a language model can

be used, which contains further information as to the probability of one recognition unit (monophone or biphone/triphone, as appropriate) following another.

For small- to medium-sized vocabularies, the word and language models are compiled into a single, integrated model. Recognition is performed using the Viterbi algorithm to find the route through this model which best explains the data. For large vocabulary systems, this approach is not viable due to the large size of the search space, and so methods of restricting its size are required. Besides the standard practice of pruning the least likely paths, this can be achieved by incorporating other information, such as data based on language usage [57] or the formation of speech, by using multiple passes, or by heuristic methods such as stack decoding [18][42].

In the third stage, word-level acoustic models are formed by concatenating the recognition units according to a pronunciation dictionary. The word models are then combined with a language model, which constrains the recogniser to recognise only valid word sequences.

The first and third stages can be performed efficiently in software (though some of the pre-processing may be better suited to a DSP). The decoding and associated observation probability calculations, however, place a particularly high load on the processor, and so it is these parts of the system that have been the subject of a number of implementations in hardware, often using custom-built chips. However, with ever more powerful programmable logic devices (PLDs) being available, such chips appear to offer an attractive alternative.

1.2.2 Speech pre-processing

Automatic speech recognition systems make use of the modulation applied by the vocal tract (throat, tongue, teeth, lips and nasal cavity); the excitation produced by the larynx is not used, even though humans infer much information from it. (Note that in a number of

Far-Eastern languages, the inflection of a syllable can profoundly affect its meaning, so requiring this information to be retained [23]).

Converting a speech waveform into a form suitable for processing by the decoder requires several stages. A typical such process [60][65] is as follows:

1. The waveform is sent through a low pass filter, typically 4 to 8 kHz. As is evidenced by the bandwidth of the telephone system being around 4 kHz, this is sufficient for comprehension.

2. The resulting waveform is sampled. Sampling theory requires a sampling rate of double the maximum frequency (so 8 to 16 kHz as appropriate).

3. The data undergoes frequency analysis using a discrete Fourier transform. This produces information about the frequency within each analysis window, which is typically 20 ms wide, with each one overlapping its neighbour by 10 ms.

4. Human hearing is not particularly sensitive to phase, so this information is removed by taking the modulus of the complex frequency data.

5. Loudness is perceived by humans on a log scale, rather than a linear one, so the log of the power is computed for the frequency data.

6. Frequency is also perceived on a non-linear scale. In particular we discriminate better between low frequency sounds than high frequency ones, and so the mel scale is used to compensate for this. A filterbank analysis is performed, whereby the frequency magnitudes are grouped into a number of bins, with the bins spaced out according to the mel scale so as to take account of our non-linear perception. Twelve such bins are typical.

7. In order to make recognition calculations less complex (specifically, to ensure that the covariance matrix is diagonal), it is required that the mel-scale filterbank com-

ponents be uncorrelated, which is not normally the case. In order to achieve this, a discrete cosine transform is effected, as a more computationally efficient approximation to principal component analysis, in order to a produce a set of mel-frequency cepstral coefficients (MFCC).

8. An additional parameter can be added in the form of an energy term, computed as the log of the signal energy.

9. Finally, further information about the "shape" of the speech data can be obtained by taking the first and second derivatives of the cepstral coefficients. Hence, starting with twelve bins, adding an energy value, and then taking the derivatives, we end up with a 39-dimensional vector.

An alternative to mel filterbank analysis is linear prediction, where the vocal tract is modelled by a transfer function, and the filter coefficients are calculated from the data in order to minimise the prediction error.

Whichever method is used, the extracted data values represent the movements of the vocal tract, and not the excitation provided by the larynx. Because the elements of the vocal tract move so slowly in comparison, the effective sampling rate is typically of the order of 100 Hz, and so one observation is produced every 10 ms.

1.2.3 HTK

HTK, the Hidden Markov Model Toolkit [60], developed by Cambridge University Engineering Department, is referred to throughout this thesis.

It consists of a suite of software tools running under UNIX, designed to facilitate the development of speech recognition applications. It can generate and train models of various different types, pre-process speech data, perform recognition, and produce accuracy figures from recogniser data. It was used for all of these functions during this research.

It was particularly useful as a benchmark against which to compare the results of the hardware and software recognisers described in this thesis, in order to verify that they were producing sensible results.

1.3 Field-programmable gate arrays

A field-programmable gate array (FPGA) is a form of programmable logic device (PLD). It typically consists of a rectangular array of configurable logic blocks (CLBs). Each CLB can contain assorted logic resources, such as look-up tables (LUTs), capable of implementing any desired boolean function; dedicated arithmetic logic, such as carry chain logic; registers, latches, shift registers, distributed memories, and so on.

The resources within a CLB can be configured as required. Similarly, the data lines that link the resources within the CLB can be configured in order to connect them together in particular ways. And the CLB array itself is immersed in a web of configurable routing, allowing the CLBs to be connected in myriad ways.

There is currently a trend towards combining fixed-function logic with reconfigurable logic, producing a so-called “system on a chip” (SoC). This started with the inclusion of blocks of dedicated RAM — themselves configurable with regard to the widths of their address and data buses — and now includes dedicated multipliers, DSP blocks, and processor cores.

In order to give the reader an idea of the numbers involved here, the FPGA used in the larger designs (XCV2000E) contains 38,400 LUTs (which can be used as any 4-input logic gate, or a 16-stage shift register, or a 16-bit RAM), the same number of flip flops (configurable as registers or level-sensitive latches), and 160 Block RAMs, each providing storage of 4,096 bits.

The field-programmable part of an FPGA comes from the fact that FPGAs can be

programmed and reprogrammed *in situ*, without having to be removed from their target PCB and placed in a chip programmer every time a new design needs to loaded, as is the case with some other types of PLD. Most FPGAs are now SRAM based, and so require a separate ROM to store their configuration data, as they are unable to retain this data themselves when switched off.

With so many resources at the designer's disposal, an FPGA provides a very powerful platform for hardware development. Its flexibility allows for all manner of complex designs; its numerous resources allow for a great deal of parallelism if the application allows it; and its ability to be reprogrammed without limitation makes it an invaluable tool for hardware development.

This is not, however, the only thing that FPGAs are good for. Making ASICs is a very expensive process, and as feature size shrinks, the cost of producing the die is increasing. The economics are such that a manufacturer needs to be expecting to ship a very large number of chips — currently of the order of hundreds of thousands for the smaller feature sizes, and continuing to rise [28][29] — before producing an ASIC becomes cost-effective. For smaller quantities, an FPGA or other PLD is cheaper.

Additionally, an FPGA's in-system programmability can be put to other uses. Unlike an ASIC, the FPGA's design can be updated after the PCB has been made and populated, and after the product has been deployed, akin to software patches being downloaded after a product has been shipped.

Taking this a stage further, one chip can be supplied with a library of designs, enabling it to perform different functions depending on the situation. For example, an FPGA could be used as part of a communications subsystem, with different configurations for different protocols, allowing hardware acceleration for all of them, but with just one chip.

Some FPGAs allow parts of the device to be reconfigured, while leaving the rest of the chip untouched. The suggestion has therefore been made for run-time reconfigura-

tion (RTR) (e.g. [22][47]), where some or all of the chip is reprogrammed at run time, providing more processing power than might otherwise be available.

Unfortunately, RTR has not been as successful as hoped, for a number of reasons. Firstly, the reconfiguration times for FPGAs, particularly the larger ones, is of the order of milliseconds, which for devices that can operate at hundreds of megahertz, is a lifetime.

To illustrate this, [17] uses RTR to update the contents of LUTs configured as ROMs, and compares this with the alternative of configuring them as RAMs instead. The authors report that while the RAM-based design has a slower clock speed and uses more resources, the LUTs can be updated much faster, by a factor of over 100.

Secondly, for partial reconfiguration, reprogramming one chunk of an FPGA affects the routing in neighbouring areas, and there is currently no obvious solution as to how to deal with that. The problem can be sidestepped by limiting the reconfiguration to replacing the contents of LUTs or RAM, or by constraining the placement of logic resources so that no routing crosses areas that will be reconfigured.

Thirdly, any complex chip relies heavily on the software that supports it, and current tools have limited support for RTR-based designs. FPGA design software continues to improve, but does still require a lot of skill of the designer — indeed, the question of whether adapting software languages in order to make it easier for software engineers to produce FPGA designs ("C-to-gates") is an ongoing debate.

Additionally, a commercial slant is mentioned in [15]: "There's no market for reconfigurability [right now]. There's a degree of reconfigurability in cellular systems, as in for changing the protocols as you move between countries, but that's a specialist area and it's done by software. The case of design reconfigurability in hardware is yet to be proved, as software is a pretty good way of achieving reconfigurability."

At present, FPGAs' power-hungry nature makes them unsuitable for mobile devices. However, once that changes, their versatility and ability to be repeatedly updated — even

if only once in a while — could see them become much more widespread than they are now.

1.4 Motivation

Speech recognition systems work best if they are allowed to adapt to a new speaker, the environment is quiet, and the user speaks relatively carefully; any deviation from this "ideal" will result in significantly increased errors. At present it is not clear whether these problems can be overcome by incremental development of the current HMM-based approach, or whether more fundamental developments are needed. In either case, it is likely that the result will place increased computing demands on the host computer. Hence, as is the case for graphics, it may be advantageous to transfer speech processing to some form of co-processor or other hardware implementation.

For most speech recognition applications, it is sufficient to produce results in real time, and software solutions that do this already exist. However, there are several scenarios that require faster recognition speeds, and so could benefit from hardware acceleration.

For example, in telephony-based call-centre applications, the speech recogniser is required to process a large number of spoken queries in parallel. If one chip could do the job of several PCs, even an expensive device could result in significant savings (not to mention taking up less space). This is typified by the AT&T system described by Gorin *et al* (1997) [12], which classifies responses to the question, "How may I help you?", in order to route calls according to their subject matter.

The possibility of saving time and money is also true of analogous non-real time applications, such as off-line transcription for dictation, where a single system would be able to process multiple speech streams at high speed.

Alternatively, the additional processing power offered by an FPGA or ASIC might be

used for real-time implementation of the “next generation” of speech recognition algorithms, which are currently in development. For example, improved recognition of fluent, conversational speech may require multiple-level acoustic models which incorporate a representation of the speech production process, for example by modelling the movement of the vocal tract. Such models are much more complex than conventional HMMs and, if successful, would inevitably lead to a substantial increase in demand for computing power for speech recognition applications.

Greater computational power can also be used to make the change from speaker dependence to speaker independence, or to make the system more robust and less sensitive to background noise.

Finally, why use an FPGA? It was originally suggested that this project make use of an FPGA with the specific intention of utilising RTR. While there has been much excitement (in academic circles, at least) that the FPGA’s unique ability to be reconfigured on the fly could be put to great use, the complexity of doing so, and the limited support of the tools, combined with the ever-increasing quantity of resources available on the device, has seen the idea pushed to one side. Instead, the FPGA’s great value has been shown in its use as a prototyping platform, either as a stepping-stone on the path to an ASIC, or as an end in itself, where an ASIC is either undesirable or uneconomical.

While this does not rule out the use of RTR at some point in the future, perhaps in order to allow the user to switch between languages or vocabularies, it remains likely that the same could be achieved by storing the corresponding data off-chip, and loading them on as necessary.

To conclude, even though processor power is always increasing, ASICs and programmable logic devices are subject to the same improvements in technology. Hence whatever we can do in software, we should be able to improve upon by using hardware.

1.5 Contribution

A PhD thesis must "represent an original contribution to knowledge, demonstrate that the student can exercise independent judgment and be worthy of publication in whole or in part in a learned journal or the equivalent."[1]

In terms of an "original contribution," it will be shown in this thesis that hardware is indeed an appropriate platform for speech recognition, with an FPGA being suitable for much of the system, the other parts being best left to alternative architectures, or to software. While it is perhaps no surprise that an FPGA is found to outperform software, it will also be shown that bandwidth and hardware resources are the principal factors affecting the speedup of hardware over software, rather than the size of the speech recognition model.

As part of the process of reaching these conclusions, what is presented here is, as far as the author is aware, the first implementation in hardware of the decoder stage of a speech recognition system, incorporating a core which overcomes bandwidth restrictions to process Gaussian mixtures for multiple speech files in parallel, as well as a Viterbi decoder core. The most closely related work [52] is described in section 3.4.

This thesis also contains the first requirements analysis for a hardware-based speech recogniser, and an original hardware implementation of the log-add algorithm for summing Gaussian mixture components in the log domain.

The log-add implementation is covered in a letter [35], with the various recogniser implementations detailed in four international conference papers [30][31][33][34], and a conference poster [32]. These all appear in the Appendix. Additionally, the requirements analysis has been submitted as a journal paper [36].

In addition to the final "products," the tools and methods used to produce them are themselves an essential part of this research. As chips grow more and more complex,

[1] University of Birmingham, Regulation 4.4.3.(3)

it becomes increasingly difficult for the design tools, and hence the designers, to utilise efficiently the wealth of resources at their disposal, and to implement, test and verify their designs within a realistic time frame.

So as a means to an end, also described here is the software toolkit that forms an integral part of this system, which — among other things — is used for benchmarking, verifying the results of the hardware, producing timing and debug information, and generating code for the FPGA designs themselves, as well as for the testbenches used in simulations.

Also worthy of mention is the hardware interface written for the FPGA development board on which the designs were implemented, which was posted on the board manufacturer's website, and has gone on to be included in the designs of other users of the hardware.

1.6 Structure of the thesis

The thesis is composed of eight chapters. Following this introduction, **Chapter 2** provides a comprehensive review of speech recognition theory, as it relates to the work described herein.

Chapter 3 is a critical review of previous speech recognition systems implemented in hardware, including the use of FPGAs, parallel processors, and serial processors, as well as current commercial speech cores and ASICs.

Chapter 4 details the designs of the various parts of the recognition system, including the parts that have been implemented in hardware, and proposed designs for those that have not.

In **Chapter 5**, the implementations themselves are discussed, focussing on issues that arose such as resource usage and speed. The system software is also described.

Departing from the main stream of the thesis, **Chapter 6** takes a look at the requirements of implementing speech recognition in hardware and software, in terms of processing time and bandwidth, and draws some general conclusions.

In **Chapter 7** are presented the results of the implementations, in the form of timing information, resource usage, and accuracy figures, along with analyses of the data.

Chapter 8 then rounds things off with a summary of findings, and conclusions.

Following this are the list of references, and also the credits page, a list of specific individuals, organisations, pieces of software, and items of hardware, which (and whom) have all played a role in this project.

Finally, the **Appendix** contains the publications produced as a result of this research.

Speech is after all only a system of gestures, having the peculiarity that each gesture produces a characteristic sound, so that it can be perceived through the ear as well as through the eye.

Robin George Collingwood (1889–1943)

2 Speech recognition theory

This chapter provides a comprehensive guide to speech recognition theory, as it relates to the implementations described later in this thesis.

Hidden Markov models (HMMs), the most commonly used basis for recognition implementations, create scope for incremental increases in the complexity of the algorithm, and hence better modelling of real speech. Presented here is the basic underlying theory, followed by a number of such extensions.

Speech recognition theory has been around for some time, and so a number of more detailed texts on the subject exist [8][44][59]. A more gentle introduction can be found in [65].

2.1 The speech recognition problem

The underlying problem of speech recognition is as follows. Given an observation sequence[1] $\mathbf{O} = \mathbf{O}_0, \mathbf{O}_1 \dots \mathbf{O}_{T-1}$, where each $\mathbf{O}_t$ is a data value representing speech which

[1] It is common in speech recognition literature to enumerate vector indices, state numbers, and time, from 1 to their respective upper limits. However, when implementing such things in hardware — and, for that

has been sampled at a fixed interval, the current time being t; and a set of potential models λ^*, each of which is a representation of a particular spoken utterance (e.g. word or sub-word unit); we would like to find the sequence of models which best describes the observation sequence, in the sense that the probability $\mathcal{P}(\lambda|\mathbf{O})$ is maximised (i.e. the probability that λ, being a subset of λ^*, is the best sequence of models given $\mathbf{O}$).

This value cannot be found directly, but can be computed via Bayes' Theorem [59]:

$$\mathcal{P}(\lambda|\mathbf{O}) = \frac{\mathcal{P}(\mathbf{O}|\lambda) \cdot \mathcal{P}(\lambda)}{\mathcal{P}(\mathbf{O})}, \tag{2.1}$$

where $\mathcal{P}(\mathbf{O}|\lambda)$ is the acoustic model probability, being the probability of the model sequence λ producing the observation sequence $\mathbf{O}$; and $\mathcal{P}(\lambda)$ is the language model probability, namely the *a priori* probability of the model sequence λ being produced (and hence the corresponding sequence of words or sub-word units being uttered).

Where $\mathbf{O}$ is a set of continuous, rather than discrete, values, $\mathcal{P}$ represents probability *density*, since the actual probability of a continuous value tends to zero. For generality, $\mathcal{P}$ is used here for both continuous probability densities and discrete probabilities.

Hence by finding the model sequence λ which maximises $\mathcal{P}(\mathbf{O}|\lambda)$, we can maximise $\mathcal{P}(\lambda|\mathbf{O})$. Since $\mathcal{P}(\mathbf{O})$ is independent of λ, this reduces to:

$$\arg\max_{\lambda} \mathcal{P}(\lambda|\mathbf{O}) = \arg\max_{\lambda} \mathcal{P}(\mathbf{O}|\lambda) \cdot \mathcal{P}(\lambda). \tag{2.2}$$

The algorithm used here subsumes the language model part, and hence $\mathcal{P}(\lambda)$, into the computations for the acoustic model probability.

The resulting recognised utterance is the one represented by the model sequence that is most likely to have produced $\mathbf{O}$. The models themselves are based on HMMs.

matter, in software — it is more efficient to number items from 0 to limit-1, and the equations presented here take this into account. The only exception to this is the current duration τ, which represents an actual quantity (the length of time spent in the current state), as opposed to an arbitrary index.

2.2 Hidden Markov models

The most widespread and successful approach to speech recognition is based on the hidden Markov model (HMM) [8][44][59], whereby a probabilistic process models spoken utterances as the outputs of finite state machines. The notation here is based on [44].

An N-state Markov model is completely defined by a set of N states forming a finite state machine, and an $N \times N$ stochastic matrix defining transitions between states, whose elements $a_{ij} = \mathcal{P}(\text{state } j \text{ at time } t | \text{state } i \text{ at time } t-1)$; these are the *transition probabilities*.

In a hidden Markov model (Fig. 2.1), each state additionally has associated with it a probability density function $b_j(\mathbf{O}_t)$ which determines the probability that state j emits a particular observation $\mathbf{O}_t$ at time t (the model is "hidden" because knowledge of the observation is not sufficient to unambiguously identify the state). The p.d.f. can be a probability density function for continuous data, or a probability distribution function for discrete data; accordingly the pre-processed speech data can be a multi-dimensional vector or one or more quantised values. $b_j(\mathbf{O}_t)$is known as the *observation probability*, and is described in more detail below.

Such a model can only generate an observation sequence $\mathbf{O} = \mathbf{O}_0, \mathbf{O}_1 \ldots \mathbf{O}_{T-1}$ via a state sequence of length T, as a state only emits one observation at each time t. The set of all such state sequences can be represented as routes through the state-time trellis shown in Fig. 2.2. The $(j,t)^{\text{th}}$ node within the trellis corresponds to the hypothesis that observation $\mathbf{O}_t$ was generated by state j. Two nodes $(i, t-1)$ and (j,t) are connected if and only if $a_{ij} > 0$.

As described above, we maximise $\mathcal{P}(\lambda|\mathbf{O})$ by maximising $\mathcal{P}(\mathbf{O}|\lambda)$. Given a state sequence $Q = q_0, q_1 \ldots q_{T-1}$ (where q_t is the state at time t), observation sequence $\mathbf{O}$, and a model λ, the joint probability of the state sequence and observation sequence, given the

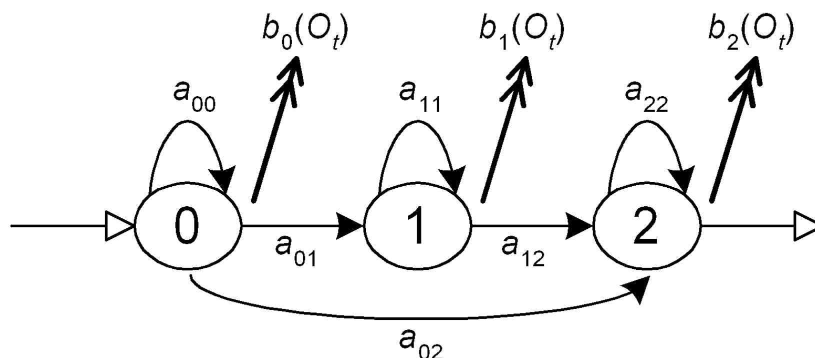

Figure 2.1: HMM finite state machine, showing the paths between states within the HMM (filled arrows), and paths between HMMs (unfilled arrows). The probability of a transition from state i to state j (transition probability a_{ij}) is shown, as is the probability of each state emitting the observation corresponding to time t (observation probability $b_j(\mathbf{O}_t)$ for each state j) (double-headed arrows)

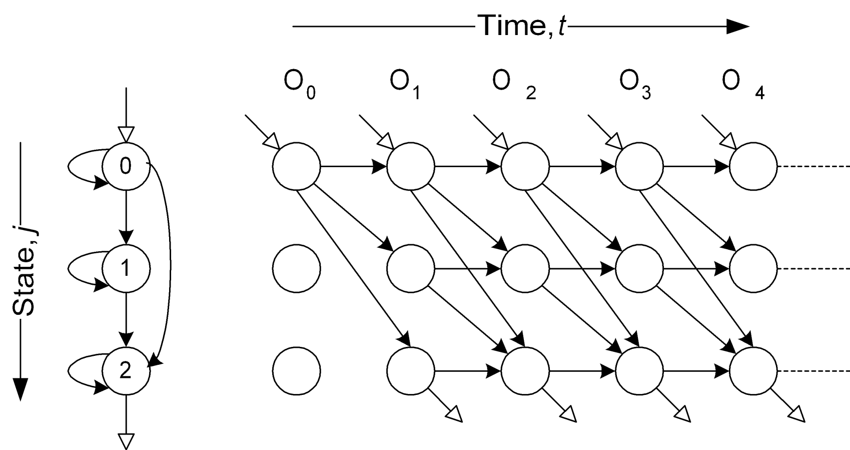

Figure 2.2: HMM trellis, showing the finite state machine for the HMM (left), the observation sequence (top), the trellis representing all possible paths between states within the HMM (filled arrows), and paths between HMMs (unfilled arrows)

model, is:

$$\mathcal{P}(\mathbf{O},Q|\lambda) = b_0(\mathbf{O}_0)\prod_{t=1}^{T-1} a_{q_{t-1}q_t} b_{q_t}(\mathbf{O}_t), \tag{2.3}$$

assuming that the HMM is in state 0 at time $t = 0$. $\mathcal{P}(\mathbf{O}|\lambda)$ is then the sum over all possible routes through the trellis, i.e.:

$$\mathcal{P}(\mathbf{O}|\lambda) = \sum_{\text{all } Q} \mathcal{P}(\mathbf{O},Q|\lambda). \tag{2.4}$$

The aim is to find the state sequence such that the joint probability of the system following that sequence, and the observation sequence being produced, given the model, is maximised. This can be computed efficiently using Viterbi decoding (below).

2.3 Viterbi decoding

Viterbi decoding [55] was first proposed in 1967 as an efficient method for the decoding of convolutional codes, an encoding system designed to prevent and correct errors when transmitting data over a noisy channel. Its use has since been extended to other applications, including gene sequencing and speech recognition.

2.3.1 Decoding

Rather than compute all possible paths though the trellis, as suggested by equation (2.4), we instead approximate $\mathcal{P}(\mathbf{O}|\lambda)$ by finding the probability associated with the state sequence which *maximises* $\mathcal{P}(\mathbf{O},Q|\lambda)$.

Firstly, we define the value $\delta_t(j)$ as the maximum probability, over all partial state sequences $q_0, q_1 \ldots q_t$ ending in state j (enumerated from 0 to $N-1$, where N is the total number of states in all HMMs) at time t (where $0 \leq t \leq T-1$), that the HMM emits the

sequence $\mathbf{O}_0, \mathbf{O}_1 \ldots \mathbf{O}_t$:

$$\delta_t(j) = \max_{q_0, q_1 \ldots q_t} \mathcal{P}(q_0, q_1 \ldots q_t; q_t = j; \mathbf{O}_0, \mathbf{O}_1 \ldots \mathbf{O}_t | \lambda). \tag{2.5}$$

It follows from equations (2.3) and (2.5) that the value of $\delta_t(j)$ can be computed iteratively as follows:

$$\delta_t(j) = \max_{0 \leq i \leq N-1} [\delta_{t-1}(i) \cdot a_{ij}] \cdot b_j(\mathbf{O}_t), \tag{2.6}$$

where i is a possible previous state (i.e. at time $t-1$).

This value determines the most likely predecessor state $\psi_t(j)$, for the current state j at time t, given by:

$$\psi_t(j) = \arg\max_{0 \leq i \leq N-1} [\delta_{t-1}(i) \cdot a_{ij}]. \tag{2.7}$$

Because we are looking at the maximum of a set of $\delta_t(j)$ values at each time frame, it is only their values relative to each other that are important, not their absolute values. As a result, we can perform operations on the data that affect the absolute values, in order to reduce the computational complexity or otherwise, as long as they maintain their position relative to each other.

2.3.2 Termination & backtracking

Each state's most likely predecessor is stored at each time frame. At the end of the observation sequence, the most likely final state q^*_{T-1} is found by simply looking for the final state whose value of $\delta_{T-1}(j)$ is highest. This value is denoted as P^*:

$$\begin{aligned} P^* &= \max_{0 \leq j \leq N-1} [\delta_{T-1}(j)] \\ q^*_{T-1} &= \arg\max_{0 \leq j \leq N-1} [\delta_{T-1}(j)]. \end{aligned} \tag{2.8}$$

Finally, in order to ascertain the most likely state sequence $Q^* = q_0^*, q_1^* \ldots q_{T-1}^*$, we trace backwards from q_{T-1}^*, looking at each state's most likely predecessor, until we reach the start of the sequence:

$$q_t^* = \psi_{t+1}(q_{t+1}^*). \tag{2.9}$$

So far, Viterbi decoding has been applied to finding the best path through an HMM representing a particular utterance. However, if we combine all of these HMMs, linking each HMM's exit state(s) to every HMM's entry state(s), one big HMM is created.

By applying Viterbi decoding to the trellis that results from this, the resulting sequence not only describes the most likely route through a particular HMM, but now, by this concatenation of HMMs, provides the most likely sequence of HMMs themselves, and hence the most likely sequence of words or sub-word units uttered.

This is known as the "one-pass" algorithm [5][6], and it allows connected speech recognition. This is distinct from continuous speech recognition, in that it assumes a finite sequence of observations, whereas the latter does not rely on the speech "ending" at a known time.

2.3.3 Assumptions

It is assumed throughout that all state sequences begin in an HMM's entry state.

The initial value of $\delta_t(j)$ often appears as:

$$\delta_0(j) = \pi_j \cdot b_j(\mathbf{O}_0), \tag{2.10}$$

where π_j is the probability of starting in state j at time $t = 0$, $\sum_j \pi_j = 1$, and $0 \leq j \leq N-1$. If no language model is used, then $\pi_j = 1/H$, where j is an entry state, and H is the number of HMMs; for non-entry states, $\pi_j = 0$. Since we are only ever interested in the relative values of $\delta_t(j)$, there is no need to multiply them all by the same constant, so this value

can be ignored. If a language model is used, this may have an effect on the allowed initial HMMs, as well as the entry states.

Is it also assumed that all state sequences end in an HMM's exit state, so when computing P^* and q^*_{T-1}, non-exit states are ignored.

2.3.4 Log-domain representation

For both software and hardware, multiplication is usually a more costly operation than addition. In particular, when designing for hardware, resource usage is a key factor, and multipliers use significantly more resources than adders. Even though some newer FPGAs have dedicated multipliers, the logic available for addition is faster and more plentiful.

Hence by performing Viterbi decoding in the log domain, all of the multiplications are converted to additions, and so can be implemented more efficiently.

Once again, the fact that it is only the relative values of $\delta_t(j)$ that are of concern, not their absolute values, means that the logarithm function, being monotonic, does not affect the validity of the result.

In addition, the log of a probability is always a negative number (assuming that we use a base greater than 1), so we negate the result. This has the effect of turning the max operation into a min; accordingly, certainty is represented as zero, while impossibility is now $+\infty$.

Hence $\delta_t(j)$ and $\psi_t(j)$ are redefined as follows:

$$\delta_t(j) = \min_{0 \leq i \leq N-1} [\delta_{t-1}(i) + a_{ij}] + b_j(\mathbf{O}_t) \qquad (2.11)$$

$$\psi_t(j) = \arg\min_{0 \leq i \leq N-1} [\delta_{t-1}(i) + a_{ij}], \qquad (2.12)$$

where $1 \leq t \leq T-1$, $0 \leq j \leq N-1$. The transition probability is similarly updated so as to be the negated log of its old value, a calculation that can be done in advance. Other

definitions are modified in the same way, and we continue in the negative log domain from this point onwards.

2.4 Language model

While the above equations can be applied to transitions both within and between HMMs, the latter are typically treated differently, in the form of a statistical language model.

Using I to represent the exit state of a previous HMM (there may be more than one exit state), and J the entry state of the current HMM, we introduce a_{IJ} as the language model probability, i.e. the probability of a transition from one HMM's exit state to another's entry state. In addition, each exit state has a probability of a transition a_I from that state to any other HMM.

Associated with the language model are two constants: s, the grammar scale factor, by which a_{IJ}, in the negative log domain, is multiplied, and p, the word insertion penalty, to which it is added. These correspond in the linear domain to a_{IJ} being raised to the power of s, and divided by p. Hence both of these values reduce the probability of a transition between utterances, thereby removing many spurious results (and the occasional correct one), and are found experimentally.[2]

Continuing in the log domain, these values are related by modifying equation (2.11) as follows:

$$\delta_t(J) = \min_{0 \leq I \leq N-1} [\delta_{t-1}(I) + a_I + s \cdot a_{IJ}] + p, \tag{2.13}$$

where $\delta_{t-1}(I) = \delta_{t-1}(i)$ for any i which is an exit state. This can be slightly optimised by grouping together the constants into one combined value dependent on I and J:

$$\delta_t(J) = \min_{0 \leq I \leq N-1} [\delta_{t-1}(I) + (a_I + s \cdot a_{IJ} + p)]. \tag{2.14}$$

[2] The notation used here is not based on any particular paper, since little or no mention is made in the literature of the equations associated with language models. The use of s and p is from HTK.

The most likely predecessor HMM, $\psi_t(J)$, is found in the same way as before, namely by replacing the 'min' by an 'arg min'.

If no language model is used, a_{IJ} is same for all HMMs, and so is equal to $1/H$ in the linear domain, or $-\ln(H)$ in the log domain, where H is the total number of HMMs (assuming each HMM has exactly one entry state). Accordingly, $\delta_t(J)$ becomes simply δ_t, being the same for *all* HMMs, so need only be computed once in each time frame:

$$\delta_t = \min_{0 \le I \le N-1} [\delta_{t-1}(I) + (a_I - s \cdot \ln(H) + p)]. \tag{2.15}$$

Conveniently, no special calculation (or in hardware, no additional computation block) is required to find the minimum (most likely) final value of $\delta_{T-1}(j)$ and its corresponding most likely final state (P^* and q^*_{T-1} respectively, as defined in equation (2.8)). Because of the requirement that the state sequence end in an HMM's final state, the most likely final state is simply equal to δ_T. In other words, the most likely final state is the exit state of the most likely predecessor HMM at time T (if the sequence were to continue to time T).

2.5 Discrete HMMs

The simplest way of determining the value of the observation probability $b_j(\mathbf{O}_t)$ is to look it up in a table (also referred to as a codebook), and that is what discrete HMMs do.

The observation feature vectors are compared to a number of candidate points in state space, and the "nearest", according to some kind of distance metric, is used to represent that observation. This is vector quantisation.

At each time frame t, the observation is quantised in this way, and for each state, the probability corresponding to the current state having produced that quantised value is looked up in a table.

This approach is computationally simpler than calculating the observation probability,

such as in the way described below, and can model any shape of probability distribution. However, vector quantisation introduces errors at an early stage in the recognition process which cannot be corrected later on, hence the tendency to use continuous HMMs nowadays.

2.6 Continuous HMMs

The use of a continuous p.d.f. seeks to overcome the problems inherent in discrete HMMs. We use a multivariate Gaussian (normal) distribution, with each state having a vector of means $\boldsymbol{\mu}_j$, and covariance matrix $\mathbf{C}_j$.

It is common to assume (or arrange) that the components of the feature vectors are mostly uncorrelated, which reduces $\mathbf{C}_j$ to being predominantly zero except along its main diagonal. This removes some of the computational complexity, leading to the following form of the Gaussian equation:

$$\mathcal{N}(\mathbf{O}_t;\boldsymbol{\mu}_j,\boldsymbol{\sigma}_j) = \prod_{l=0}^{L-1} \frac{1}{\sigma_{jl}\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{O_{tl}-\mu_{jl}}{\sigma_{jl}}\right)^2\right), \quad (2.16)$$

where $\mathbf{O}_t$ is the vector of observation values at time t; $\boldsymbol{\mu}_j$ and $\boldsymbol{\sigma}_j$ are mean and variance vectors respectively for state j; O_{tl}, μ_{jl} and σ_{jl} are the elements of the aforementioned vectors, enumerated from 0 to $L-1$.

This model is still limited by the fact that observed distributions are not Gaussian-shaped, so a further extension is to use multiple-component Gaussian mixtures. For this, the final probability is the sum of a number of individually weighted Gaussian distributions:

$$b_j(\mathbf{O}_t) = \sum_{m=0}^{M-1} c_{jm} \cdot \mathcal{N}(\mathbf{O}_t;\boldsymbol{\mu}_{jm},\boldsymbol{\sigma}_{jm}), \quad (2.17)$$

where c_{jm} is the mixture weight for state j and mixture component m, the mixture com-

ponents are enumerated from 0 to $M-1$, and:

$$c_{jm} \geq 0, \quad \sum_{m=0}^{M-1} c_{jm} = 1. \tag{2.18}$$

As with Viterbi decoding, we can reduce the computational complexity by performing the necessary calculations in the negative log domain, resulting in the following equation for each mixture component:

$$-\ln(c_{jm} \cdot \mathcal{N}(\mathbf{O}_t; \boldsymbol{\mu}_{jm}, \boldsymbol{\sigma}_{jm})) =$$

$$\left[-\ln(c_{jm}) + \frac{L}{2}\ln(2\pi) + \sum_{l=0}^{L-1} \ln(\sigma_{jml})\right] + \sum_{l=0}^{L-1} (O_{tl} - \mu_{jml})^2 \cdot \left[\frac{1}{2\sigma_{jml}^2}\right]. \tag{2.19}$$

For the implementation that uses a simple Gaussian, c_{jm} is set to 1, hence $\ln(c_{jm})$ disappears from the equation.

Note that the values in square brackets are dependent only on the current state, not the current observation, so can be computed in advance. This reduces the equation to its final form:

$$-\ln(c_{jm} \cdot \mathcal{N}(\mathbf{O}_t; \boldsymbol{\mu}_{jm}, \boldsymbol{\sigma}_{jm})) = S_{jm} + \sum_{l=0}^{L-1} (O_{tl} - \mu_{jml})^2 \cdot V_{jml}, \tag{2.20}$$

where S_{jm} and V_{jml} replace the bracketed terms from above.

For computation in hardware, S_{jm} can be subsumed into the summation by adding an extra element to all of the vectors, assigning this value to V_{jml}, and setting O_{tl} to 1 and μ_{jml} to 0. In software, we can keep a running total which is intialised to S_{jm}.

The result of this is that for each vector element of each state, we now require a subtraction, a square and a multiplication.

2.7 Gaussian mixture summation (log-add algorithm)

When it comes to summing the mixture components (equation (2.17)), we are faced with the problem of performing addition in the linear domain of values computed in the log domain.

While we could use look-up tables or CORDIC to convert between domains (as covered in section 3.7), a convenient algorithm exists for this specific problem [14]. It removes the need to convert between domains at all, instead relying on a look-up table significantly smaller than for the logarithm or exponential operations, along with some simple arithmetic computations — and hence well suited for implementation on an FPGA.

Given two values $\ln(A)$ and $\ln(B)$ for which we would like to compute $\ln(A+B)$:

$$\begin{aligned} A+B &= A(1+B/A) \\ \Rightarrow \ln(A+B) &= \ln(A(1+B/A)) \\ \Rightarrow \ln(A+B) &= \ln(A)+\ln(1+B/A). \end{aligned} \tag{2.21}$$

To compute the result, we work out $\ln(B/A)$, which is equal to $\ln(B)-\ln(A)$, and then use a look-up table to map that value to $\ln(1+B/A)$. Since the values in this table are dependent on the *relative* values of A and B, it can be smaller than one which relies on their actual values (see also section 4.5).

In order to minimise the size of the look-up table without compromising accuracy, we require that $A \geq B$, hence limiting $1+B/A$ to the range $[1,2]$, and switch the values if this condition is not met.

2.8 Duration modelling [25][37][44]

In the form described above, the probability of the system staying in state j for a duration τ forms a geometric sequence, equal (in the linear domain) to $a_{jj}^{\tau-1}(1-a_{jj})$. However, this has not been found to model observed speech accurately. This can be improved on by replacing this implicit duration model with an explicit one $d_j(\tau)$.

Accordingly, $\delta_t(j)$ is redefined (still in the negative log domain), with the constraint that the state at time $t+1$ does not equal the state at time t, as follows:

$$\delta_t(j) = \min_{\substack{0\le i\le N-1 \\ i\ne j}} \min_{1\le\tau\le D} [\nu_t(i,j,\tau)], \tag{2.22}$$

where τ is the number of times that we remain in state j, i is the state at time $t-\tau$, D is the maximum duration, and:

$$\nu_t(i,j,\tau) = \delta_{t-\tau}(i) + a_{ij} + d_j(\tau) + \sum_{s=0}^{\tau-1} b_j(\mathbf{O}_{t-s}). \tag{2.23}$$

$\nu_t(i,j,\tau)$ is the probability at time t, of the system having moved to state j at time $t-\tau+1$ from state i at time $t-\tau$, and then having stayed in state j for duration τ. Note that, because self-transitions are now handled explicitly, cases where $i=j$ (i.e. where the current and previous states are the same) are discounted. This can equally be effected by setting a_{jj} to zero ($+\infty$ in the negative log domain).

As will become apparent in section 4.6, for more efficient implementation in hardware and software, it is convenient to extend these definitions. Firstly, if we take the minimum of $\nu_t(i,j,\tau)$ over all previous states, we can group the terms into those that are dependent on the previous state, and those that are not:

$$\min_{\substack{0\le i\le N-1 \\ i\ne j}} [\nu_t(i,j,\tau)] = \left[d_j(\tau) + \sum_{s=0}^{\tau-1} b_j(\mathbf{O}_{t-s})\right] + \min_{\substack{0\le i\le N-1 \\ i\ne j}} \left[\delta_{t-\tau}(i) + a_{ij}\right]. \tag{2.24}$$

Secondly, by removing $d_j(\tau)$, we can define $\xi_t(j,\tau)$ as the probability at time t, of the system having moved from its most likely predecessor state at time $t-\tau$ to state j at time $t-\tau+1$, and emitting the observation sequence from time $t-\tau+1$ to t:

$$\xi_t(j,\tau) = \sum_{s=0}^{\tau-1} b_j(\mathbf{O}_{t-s}) + \min_{\substack{0\le i\le N-1 \\ i\ne j}} \left[\delta_{t-\tau}(i) + a_{ij}\right]. \tag{2.25}$$

This value is useful, as it can also be defined iteratively, avoiding the need for any recomputation:

$$\xi_t(j,\tau) = \xi_{t-1}(j,\tau-1) + b_j(\mathbf{O}_t). \tag{2.26}$$

Finally, we can redefine $\delta_t(j)$ in terms of $\xi_t(j,\tau)$:

$$\delta_t(j) = \min_{1\le\tau\le D}[d_j(\tau) + \xi_t(j,\tau)]. \tag{2.27}$$

The way in which these values fit together is illustrated in Fig. 4.10 on page 62.

2.9 Summary

In this chapter, the aspects of speech recognition theory of relevance to this research have been presented. The underlying problem has been posed, and hidden Markov models introduced to provide a solution, with Viterbi decoding enabling them to do so. Discrete and continuous HMMs have been described, along with a number of extensions to the basic algorithm, namely the language model, Gaussian mixtures, and duration modelling.

Before looking at how this assortment of algorithms might be implemented in hardware, it is first necessary to cast a critical eye over previous such implementations, and so that is the area visited next.

The newest computer can merely compound, at speed, the oldest problem in the relations between human beings, and in the end the communicator will be confronted with the old problem, of what to say and how to say it.

Edward R. Murrow (1909–1965)

3 Speech recognition in hardware

It is only in the last few years that desktop PCs have been powerful enough to allow large-vocabulary continuous speech recognition to be performed, in real time, in software. At present though, for best results, they still rely on being trained to recognise one speaker, with minimal background noise. Even then, steps have to be taken in order to reduce the computational complexity so that real time recognition is feasible.

Before this was possible, or when it was necessary to try out more complex algorithms, only hardware had the computational resources to achieve this.

Initially, hardware implementations tended to be based on parallel arrays of one kind or another, often using custom chips. As the technology has improved, the focus has shifted towards serial implementations, making use once again of custom chips, or microcontrollers or DSPs. Since the appearance of the FPGA, that too has been used as an experimental platform.

Of the three principal stages of the speech recognition process, it is the decoding part that takes centre stage in hardware implementations. Pre-processing tends to be performed in software, or left to a DSP (though Gómez-Cipriano *et al* (2001) [11] use an FPGA for

feature extraction). Backtracking, as described below, is much better suited for processing in software, rather than hardware.

So it is Viterbi decoding, and latterly, observation probability computation, that are by far the most popular choice for implementations in hardware, and it is an assortment of such implementations that is reviewed in this chapter.

3.1 Accuracy measures

The accuracy figures which appear here are intended to give an indication of how good the cited recognition systems are. However, comparing values for different recognisers can be misleading, as there are a number of factors which can affect this value.

First is the type of recogniser. A small-vocabulary system, such as one used for digit recognition, has a smaller number of HMMs, with fewer parameters for the training process to set, and a correspondingly constrained search space. So *for its specified vocabulary*, it is likely to achieve a high score; for anything outside this, it will not. This contrasts with a large-vocabulary recogniser, which may give a lower recognition rate, but is capable of recognising many more words.

In a similar vein, a speaker-dependent system is likely to perform well for the speaker it has been trained for, and not so well otherwise, whereas a speaker-independent system is not tied to any one speaker.

The type of recogniser also places limits on recognition rate. Generally speaking, the better the model is at representing speech, the more processing that is required, and the higher the recognition rate is likely to be. So continuous HMMs tend to do better then discrete ones, and biphone & triphone models tend to outperform monophone models.

Finally, the accuracy rate can be manipulated by what is and is not included in that figure. The value used by HTK in section 7.3 takes both correctly identified recognition

units and incorrect insertions into account, resulting in a lower score than one which relies on correct utterances alone. This value can be further increased by treating similar-sounding units as being the same unit.

3.2 Parallel systems

Parallel implementations of speech recognition systems have been produced before, most using HMMs. In contrast to the approach described here, previous implementations have generally used multiple processing elements (PEs) of varying sophistication, either at the board or ASIC level, rather than a programmable logic device.

3.2.1 Custom ICs

Murveit *et al* (1989) [39] use 5 custom integrated circuits (ICs) to implement a system whose Viterbi decoder has much in common with the designs presented in this thesis. Their decoder chip has 3 parallel adders to add $\delta_t(j)$ and a_{ij}, a comparator to find the minimum, and another adder to add $b_j(\mathbf{O}_t)$. It also incorporates a scaler. Another IC handles the language model, with the model probabilities, and $\delta_t(j)$ and $\delta_{t-1}(i)$ stored in off-chip RAM.

The system is designed for continuous real-time speech, with a vocabulary of 3,000 words (9,000 states), based on bigrams and discrete HMMs. As is described in chapter 5 with reference to the implementations presented there, memory bandwidth is the major performance-limiting factor.

This work is continued by Stölzle *et al* (1991) [50], using 50,000 states and discrete HMMs. Again, a dedicated Viterbi decoder chip is used, accompanied now by a back-track processor, which computes $\psi_t(j)$ values for each state while the corresponding $\delta_t(j)$ values are being computed.

It is pointed out in [37] that custom Viterbi decoders like these have two disadvantages. Firstly, pruning is made harder, as having multiple computations done in parallel means that for any time to be saved, *all* of the processing blocks' data must be pruned equally. If this does not happen, the other blocks must wait for the unpruned data to finish being processed.

Secondly, such decoders have less flexibility when it comes to interconnects. Both of these implementations assume that most states have up to three predecessors, and that only left-right transitions (i.e. transitions, within an HMM, that go from a state to itself, or to a state with a higher index) are allowed, thereby limiting themselves to one class of model.

In contrast, the implementations described in chapter 5 allow transitions in both directions, which is useful for silence models in particular. Also, although the underlying design assumes that states have three predecessors (four in the case of entry states), this could be changed at compile time if necessary.

3.2.2 SIMD arrays

Many of the first attempts at using hardware for speech recognition focussed on the single instruction, multiple data (SIMD) array, whereby all the processing elements (PEs) in the array carry out the same instructions on different data.

SIMD arrays are not directly comparable to FPGAs, since FPGAs do not contain processing elements as such. However, the way in which the algorithm is implemented on a SIMD array, in particular, the extent to which certain tasks can be pipelined and/or run concurrently, can be applied when implementing them on an FPGA.

Bisiani *et al* (1989) [3] use 3 general-purpose processors, with local and shared memory, to perform Viterbi decoding, splitting the 1,000 words in the model between them, and averaging a speed of 1.3 times real time.

They point out that while each PE is free to process states at its own pace, they must all finish processing the data for the current time frame before the next set of data can be loaded. Hence balancing the load between PEs is important in order to avoid them sitting idle, and this is an issue that comes up in a number of SIMD implementations. In this case, this is achieved by giving each processor its own data queue, but with a mechanism to allow data to be redistributed if an imbalance occurs.

The INMOS T800 transputer has proven to be a popular choice for SIMD implementations, with load balancing again a highly relevant issue. Alexandres *et al* (1990) [1] compare different topologies (linear, ring and tree) of PEs in a processor farm, whereby data values are sent by the host through the PEs in sequence, until a free one is found; the results of the processing are sent back through the network to the host. The load balancing mechanism is therefore distributed and automatic.

Pruning is also addressed: the host PE compares all the data to a threshold, and only sends values above the threshold to the processor farm. The authors report that this does not affect accuracy, while improving the processing speed by a factor of 3 to 4, or allowing fewer processors to be used.

The implementation uses discrete HMMs, and a vocabulary of 1,000 words.

Sutherland *et al* (1990) [53] eschew this fine-grained approach in favour of a more coarse-grained, distributed model, where each PE is given its own set of data to work on at the start of every time frame, with the load balanced prior to run time. The host PE's task is reduced to broadcasting the quantised observation value to the other PEs, gathering the backtracking information, and collecting the computed probabilities in order to perform pruning. This design operates on 4 transputers in real time.

As an aside, it is interesting to note that L.S. Lee *et al* (1991) [23] have implemented a recognition system, also using the INMOS T800, for the recognition of Mandarin Chinese. Unlike European languages, in Mandarin the inflection (tone) of each syllable has as much

effect on its meaning as the vowels and consonants. As a result, the tonal information, normally discarded when the feature vectors are extracted from the speech waveform, must instead be separated, processed in order to ascertain which of Mandarin's four tones has been used at any particular time frame, and then combined with the output of the Viterbi decoder.

Contrasting with these transputer implementations, S.W. Lee *et al* (1992) [24] use an "orthogonal multiprocessor," whereby n processors have access to row or column buses connected to an $n \times n$ array of memory blocks. This array then mirrors the HMM trellis, with each row assigned to a state, and each column a time, the values wrapping around when the edge of the array is reached. As many $\delta_t(j)$ values are updated in parallel as there are PEs.

Finally, a particularly relevant work concerning recognition on a SIMD array is that of Mitchell *et al* (1995) [37]. They employ the MasPar MP-1, a 128×128 array of PEs, each one consisting of a 4-bit ALU with 16 Kb of local memory. This is used to perform recognition using continuous HMMs, and incorporating duration modelling, taking maximum advantage of its inherent parallelism.

The PEs are arranged in an array, with each row assigned to a state. The observation probabilities are calculated first, by assigning each PE in a row to one element of a multivariate Gaussian distribution. The calculations are done in the linear domain, with the values for all the states being summed and exponentiated in parallel.

For the Viterbi decoding, each PE in a row corresponds to a duration τ, up to the maximum D, with the values scanned across each row in order to find the $\delta_t(j)$ values, computed in the log domain.

This implementation, using 64 monophones of 3 states each, shows a speedup over software of more than an order of magnitude. Increasing D slows the system down, though not by much (the time taken for the test increases from 0.262 s to 0.569 s as D increases

from 4 to 32, compared to 5.35 s and 28.18 s respectively for the software version). A similar effect is shown when D is maintained at 32, but the number of monophones is varied.

This form of distributed processing, which is used for training as well as recognition, clearly demonstrates that parts of the speech recognition algorithm have the potential to be parallelised effectively, leading to significant speedup over software.

3.2.3 MIMD arrays

Kimball *et al* (1987) [20] implement their speech recogniser on a multiple instruction, multiple data (MIMD) array. The Butterfly Parallel Processor's PEs each contain a microcontroller, local RAM which is also accessible to all other PEs, a communications co-processor, and other computing and management hardware.

Whereas a SIMD implementation farms out data to PEs as they become free, with each PE performing the same operations, a MIMD array farms out tasks as well as data.

With 97 PEs, processor utilisation was initially 35%. This was increased to 79% by reordering, in advance, certain tasks that were independent of each other, to avoid delays due to data dependence.

In their initial implementation, all of the calculations for time t, including scaling, were calculated before any processing for time $t+1$ was started. This had the result that many PEs stood idle while others finished their tasks. Two optimisations were made in order to overcome this. Firstly, scaling was delayed by one or more time frames, allowing the maximum at time t to be computed even after some PEs had started dealing with data for time $t+1$.

Secondly, the "word starting score," i.e. the between-HMM probability used in the absence of a language model, presumably calculated as in equation (2.15), is also dependent on all data from the previous time frame. Idling time was minimised by processing entry

states *after* all the non-entry states, allowing any processing from the previous time frame to be completed.

The problem of finding the global maximum was also mentioned. Rather than having each PE compare its value to a single globally-accessible maximum, leading to bus contention, a binary tree structure was used, with local maxima propagating up through the tree's nodes.

The system was tested with a 335-word speaker-dependent model, based on discrete HMMs, with no language model, and achieved a recognition rate of 90%.

3.2.4 Associative string processors

Krikelis (1989) [21] makes use of an associative string processor for Viterbi decoding for speech recognition. The system consists of a number of parallel "substrings," each containing a number of associative processing elements, and connected to each other via a communications network and related logic.

What makes this platform associative, and hence distinct from other parallel architectures, is that the simple PEs are addressed by their activity and data content, rather than by address or identifier, not unlike the way in which a location in a content-addressable memory is referenced by the data it contains.

The HMM states are distributed amongst the PEs, two per state, and processed in parallel. As a result of using very simple PEs, and reduced inter-processor communication arising from their associative nature, devices with up 8,000 processors are reported to be feasible.

3.3 Serial systems

In contrast to the parallel designs described above, the increased processing power now offered by processors and ASICs — not to mention the lower cost — has led to a shift towards such devices.

Shozakai (1999) [49] uses an ASIC containing a DSP core for feature extraction and Gaussian computations, and a RISC microprocessor core for the Viterbi decoding. Tied-mixture Gaussian mixtures are used, with 54 Japanese monophone HMMs.

Based on this monophone model, the system was tested using speech in five languages, with a vocabulary of 128 words, and speaker dependence. The accuracy under these conditions is generally upwards of 90%. With Japanese speech, accuracy averages 97%, decreasing to 92% when background noise is added (the system is designed for use in cars).

Nakamura *et al* (2001) [40] describe an embedded system incorporating an ASIC which also performs observation feature extraction and Viterbi decoding. Discrete HMMs of 5 states each, representing 64 monophones, are used. An FPGA is used for training.

The authors report that the hardware, running at 17 MHz, can perform recognition in real time. They add that if their ASIC were operating at the same speed as the processor used for testing equivalent software (Pentium III 750), the ASIC would be 5.3 times faster.

In contrast to this, Shi *et al* (2001) [48] employ an ASIC containing an 8051 core for almost all the processing, including feature extracting, with only the minimum of support logic (mainly for analogue-digital conversion). The rationale of not using a DSP core is that they are expensive in comparison — but the trade-off is the reduced processing power available.

Dynamic time warping is used (see also below), and the system performs both training and recognition. The authors state that the chip is capable of accuracy above 90% for a constrained vocabulary.

What is clear from these implementations is that, although a system-on-chip design can do speech recognition, current ones have only enough processing power to cope with small vocabularies and the simpler types of model (e.g. discrete, tied mixture, etc).

3.4 FPGAs

Providing a compromise between the processing power of hardware and the flexibility of software, the emergence of FPGAs in the 1990s provided a new platform for the development of speech recognition systems.

Schmit & Thomas (1995) [46] present an early FPGA implementation of an HMM-based application, on a Xilinx 4000-series device. In this case, they use Viterbi decoding to correct errors made by a person typing, resulting in a system 25 times faster than equivalent software.

Although this design is not entirely comparable to those described in chapter 4, the authors observe that beam searching can double the speed of the software without significantly reducing its accuracy. They feel, however, that comparing all the values to a threshold in hardware would take as long as doing the computation for that state — an observation borne out in the designs described later, though in that case it affects the latency (number of clock cycles between data entering a pipeline and leaving it, with data nonetheless produced at the rate of one item per clock cycle), not the delay (number of clock cycles between data items being produced), of the system.

In addition, the authors state that maintaining the data structure used in software for recording pruned states "destroys the regularity and simplicity of the algorithm which made it especially amenable to hardware implementation." Again, in the designs presented later, despite using a scaler (section 4.3.2) which effectively prunes the least likely states, it was found to be simpler to mark pruned states by giving them impossible proba-

bilities, than to remove them from the data stream.

Vargas *et al* (2001) [54] use two Altera FPGAs to implement a simple isolated word recognition system. The model uses up to 10 words, with 6 states per discrete HMM, and 128 codebook entries per state. They take advantage of parallelism within the Viterbi algorithm to achieve a speedup over software of the order of 500 times, with accuracies for this task approaching 100%.

A novel implementation is demonstrated by Jou *et al* (2001) [19], who propose an "efficient VLSI architecture," prototyped on an FPGA. It takes advantage of the left-right nature of HMM state machines used in speech recognition by merging every four columns of the Viterbi trellis into one.

The authors state that this approach saves on time and resources. While this could be useful for faster-than-real-time transcription, there is likely to be little gain when processing real-time speech, as the system would have to wait for the same amount of time between new observations whether it was processing one or four at a time.

The implementation uses Gaussian mixtures, and computes each component's value in a similar manner to that described in section 4.4. However, rather than using log-add, they pick the most likely component.

The work most closely related to the research described in this thesis is that done by Stogiannos *et al* (2000) [52], based on Stogiannos (1999) [51]. They use discrete-mixture HMMs, in which the elements of the observation vector are quantised in advance, allowing the probability associated with each element to be looked up in an off-chip codebook, rather than calculated. These values (in the log domain) are then summed, converted to the linear domain using another look-up, and further summation takes place (as for Gaussian mixtures). The conversion back to the log domain and the Viterbi decoding are performed in software.

This approach uses a lot of external RAM: 64 Mb of SDRAM for the codebook val-

ues, and 512 Kb of SRAM for the domain conversion (organised as four 128 Kb look-up tables). In contrast, all but one of the designs described later use continuous probability distributions, and so compute the mixture components on the FPGA. Use of an alternative algorithm removes the need for a domain conversion for the mixture component summation, greatly reducing the large storage and bandwidth requirements inherent in a RAM-based implementation. In addition, the Viterbi decoding is performed in hardware. In all cases, the designs take advantage of more recent devices which are faster and have more resources available.

Their speech model uses 10,900 states grouped into 1,100 "genones" (groups of states with similar properties), with each genone being represented by 32 Gaussian mixture components, and each vector containing 15 elements. The model is described as being capable of recognition accuracy above 85% for a vocabulary of 1,500 words. The system is designed for an Altera FLEX 10KE running at 66 MHz.

3.5 Commercial products

A small number of commercial speech recognition ASICs exist, such as Sensory's RSC-300/RSC-364 and RSC4x family [67], which use a RISC microprocessor with a neural network; their Voice Direct 364, which is also based on a neural network; and Philips' SBF1005 HelloIC [66], which is based on a DSP. All three are designed for applications requiring a small vocabulary (typically 60 words or less), and boast a speaker-independent recognition accuracy of 97% or more. (Further performance comparisons are not possible due to a lack of suitable information).

Although, for the time being, recognition chips and IP cores only handle small vocabularies, their prevalence in toys, automotive applications, and mobile phones suggests that the market for such devices in embedded and mobile systems will continue to increase

[10][38].

As regards FPGAs, there are no cores designed specifically for speech recognition. However, cores do exist for performing Viterbi decoding for signal processing, such as those produced by TILAB [69] and Xilinx [71]. In addition, some DSPs have dedicated logic for Viterbi decoding, for example, the Texas Instruments [68] TMS320C6416, and the TMS320C54x family.

In both cases, however, these decoders are designed for signal processing applications, which, as outlined below, have different requirements from speech recognition, including narrower data widths, different data formats, and fewer states.

3.6 Alternative recognition methods

The hidden Markov model is by far the dominant underlying algorithm used in speech recognition systems, both commercially and in research. However, there are alternatives which provide a useful comparison.

Dynamic time warping[1] (DTW) [8] predates HMMs, and is in fact a special case of HMMs. It works by comparing two utterances, stretching or compressing one (hence "warping") in order to try and match it to the other. The degree to which the utterance is warped determines a value, not unlike the $\delta_t(j)$ used by HMMs, though computed without transition probabilities, and with observation probabilities replaced by a distance metric (typically Euclidean or "Manhattan"). This value must be minimised in order to find the most likely match.

DTW was superseded by HMMs because the former provides less flexibility, as it cannot be made more robust by training on large amounts of data. Conversely, it has a use where data is limited, as a single utterance can be used as a template in lieu of training data.

[1]The author is not aware of any connection between this and the Rocky Horror Show.

Its relative simplicity was of use when implemented by Shi *et al* (2001) [48], as described above.

Also mentioned earlier were neural networks. Rather than use any particular algorithm, a neural network is trained on a set of template patterns (e.g. a set of words used for command and control application). It is then sent data extracted from incoming speech, and the data is compared to the templates. The neural net picks the most likely template, or selects a number of most likely candidates, with a final one being chosen after further processing.

Neural nets are simple to train, but their pattern-matching abilities are limited. They are suitable for recognising a small number of isolated words, but they cannot cope with large-vocabulary continuous speech. Their inherent parallelism, however, does make them suitable for implementations in hardware, such as the FPGA version described by Eldredge & Hutchings (1994) [9]. A more general approach is presented by Chen & Jamieson (1996) [7].

Finally, a more unusual approach is introduced by Bohez & Senevirathne (2001) [4]. They use fractals for clustering phonemes, and report that this method is good for endpoint detection and segmentation, but not dealing with whole words. It is suggested that this method on its own is not suitable for recognition, but could be used in conjunction with other techniques.

3.7 Gaussian mixture summation

There do not appear to be any published hardware implementations of the log-add algorithm described in section 2.7. The alternative method is to convert the data from the log domain to the linear domain (i.e. take the exponential), perform the summation, and then take the log of the result.

Two different approaches have been previously implemented on an FPGA. In [52], a RAM-based look-up table is used to perform the conversion on 16-bit log-domain values, producing 16-bit results; 128 Kb of storage is required for this. The reverse operation is performed in software; were it to be implemented in hardware, it seems likely that another similarly sized table would be required. This compares to one 11 Kb look-up table in the design described in section 4.5.

Alternatively, CORDIC [2][56] provides a very resource-efficient method of performing non-linear operations like these. An iterative implementation requires a very small look-up table (just one entry per bit of accuracy), and incurs a delay of one clock cycle per bit. A fully pipelined version does not use a look-up table at all, and incurs a latency of one clock cycle per bit. Two CORDIC blocks would be required for the two conversions, with a couple of additional cycles for the summation itself. The log-add design presented later, using 24-bit numbers, has a total latency of 4 cycles.

3.8 Other hardware implementations

Although the choice was made to focus on particular algorithms for realisation in programmable logic, there are a number of other methods, and other parts of the recognition process, which have also been implemented in hardware. A selection are presented below.

3.8.1 Convolutional decoding

Viterbi decoding [55] was originally developed as a more efficient method for the decoding of convolutional codes. When sending data over a noisy channel, the data is encoded using a finite state machine (FSM) in such a way that the extra bits added to the bitstream can be used by the receiver to extract the original information, even if the bitstream has errors in it.

The size of the FSM used for the encoding is dictated by the constraint length K, where K is either the number of previous inputs stored, or that value plus one in order to include the current input. Hence the number of states in the FCM is either 2^K or 2^{K-1}, respectively, depending on which definition is used.

This FSM is also used for the decoding, with a Viterbi trellis used to find the most likely path. Each transition corresponds to a possible bit sequence of length R (typically 2, 3 or 4). Instead of transition or observation probabilities, an error metric is computed which measures the "distance" between the transition's associated R-bit sequence, and the next R bits actually received; this is typically the Hamming distance, i.e. the number of bits which differ between the two values, or the Euclidean distance. The most likely path is the one with with the smallest total error metric.

Besides the lack of probabilities, this also differs from Viterbi decoding for speech in the nature of its state machine. In some of the implementations described later, 634 biphones and triphones of 3 states each are used, which together form a machine of 1,902 states, though with the number of allowed transitions constrained.

For convolutional encoding, the number of states is dependent on K. As an example, the Texas Instruments TMS320C6416 allows a value of K between 5 and 9, hence 16 to 256 states, with the distance metric based on Euclidean distances.

Aside from the aforementioned commercial FPGA cores, Yeh *et al* (1996) [58] implement a Viterbi decoder with $K = 14$. This mammoth decoder is implemented using 36 Xilinx XCV4010 FPGAs, spread across 7 boards with a custom backplane, and utilising a multi-ring topology. The reconfigurable nature of the FPGAs is used to allow smaller decoders to be implemented using the same hardware.

Then, as now, one of the key points is the use of FPGAs as a more cost-effective solution for low-volume applications, though at the expense of lower processing speeds than ASICs.

3.8.2 Traceback

Traceback/backtracking for convolutional decoding has also been implemented in hardware (e.g. Lin (2000) [26]). The traceback depth is fixed by the amount of logic implemented, and it has been shown [27] that it is sufficient to set this equal to $5K$, where K is the constraint length mentioned above.

For convolutional decoding, K, and hence the number of states, is sufficiently small to make a hardware backtracker feasible. For speech recognition, however, 1,900 states corresponds to a nominal K value of 12, requiring predecessor information for all of those states to be stored for up to 60 $(= 5 \times 12)$ time frames. This would require more memory than is likely to be available on an FPGA or ASIC, and of course, a more complex speech model would need even more space.

3.8.3 Training

Training the speech models for subsequent use in recognition is more computationally demanding than the recognition itself, not least because it cannot be performed in the log domain, hence requiring rather more than additions and comparisons. This, however, is balanced by the fact that it does not need to be performed in real time.

A number of hardware implementations exist. Yun *et al* (1997) [61] present a reconfigurable parallel processor, with a small number of PEs each containing a RISC processor and an FPGA.

Pepper *et al* (1990) [43] utilise a ring of parallel processors arranged as a skewed SIMD array, with data passing in both directions around the ring. The authors report that this approach is "optimally efficient" and minimises interprocessor communications.

In addition, [24][37][48], whose architectures are described above, also contain details of training algorithms applied to hardware.

3.8.4 Trees

Roe *et al* (1989) [45] use ASPEN, a tree-structured parallel computer, for two purposes. The first is level-based pruning, where all but the K best nodes are pruned, as opposed to the normal method, whereby those nodes whose associated probability falls below a threshold are pruned. The second is to implement pattern recognition, using finite state machines to model grammars.

The PEs, each consisting of a DSP plus local memory, are arranged in a binary tree formation. Using 127 PEs, the authors predict that a 1000-word vocabulary, with continuous HMMs, could be processed in real time — which, at the time of writing, would have been a significant speedup over software.

The key advantage of the tree-based architecture is the simplified, hierarchical, interprocessor communication, which they make use of in their algorithms.

3.9 Summary

This chapter has taken a look at previous hardware implementations of speech recognition systems. After commenting on ways in which accuracy is measured, a number of types of parallel system have been considered, along with serial systems, and naturally FPGAs as well. Some commercial recognition products have also been mentioned.

Hardware implementations of recognition methods besides HMMs have been described, as have alternatives to the log-add algorithm. Finally, an overview has been given of a few implementations in areas indirectly related to recognition.

Having looked at how parts of the recognition process have been implemented before, it is now time to propose new designs, inspired by these, and based on the theory described in the previous chapter.

Good design keeps the user happy, the manufacturer in the black and the aesthete unoffended.

Raymond Loewy (1893–1986)

4 System design

In this chapter, the designs for the various parts of the speech recognition system are presented. Because the world of electronics moves at a pace which ensures that any device-specific designs will soon be obsolete, those described here are independent of any particular chip. Instead, they assume a hardware platform with, as a minimum, the resources of today's FPGAs.

The debate over how to partition a system between hardware and software is an ongoing one. In this case, we have the speech pre-processing, recognition, and backtracking stages to implement. As mentioned previously, the pre-processing can be done using dedicated DSPs, or in software; the backtracking process requires large amounts of data storage, and indexing operations, for which software is better suited. It is the recognition part, including Viterbi decoding, and in particular the computation of observation probabilities, which requires significant number crunching, and for which no suitable device currently exists — and so it is this part that has been the subject of this research, and for which the hardware designs are presented here.

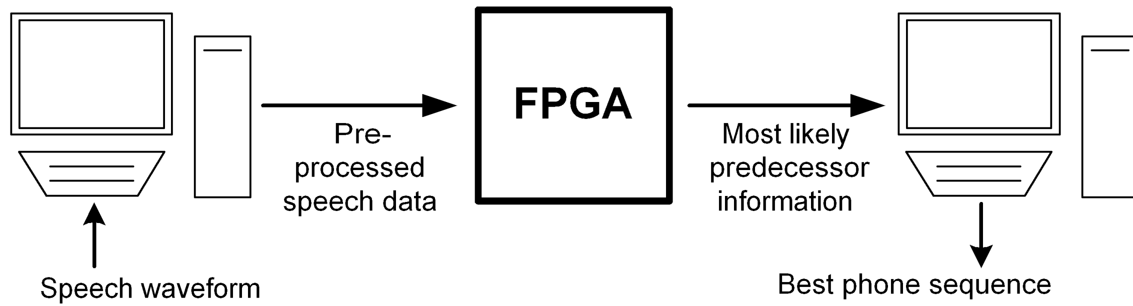

Figure 4.1: System structure

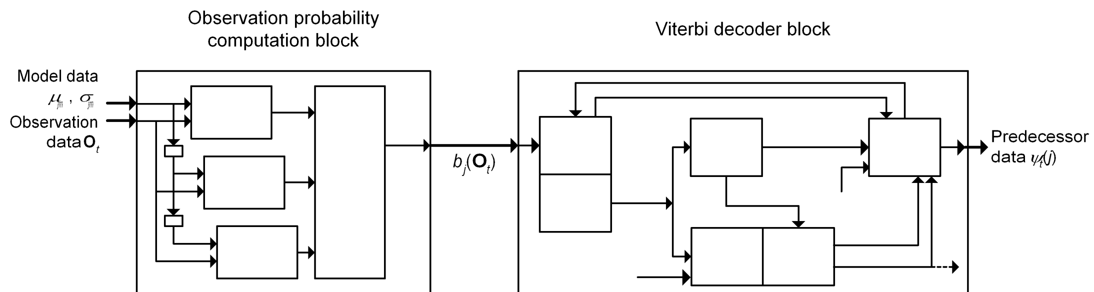

Figure 4.2: Recogniser structure

4.1 Structure

The design assumes that the hardware recogniser will act like a co-processor within a computer, with the host passing the pre-processed speech and model data to the recogniser, and the recogniser sending the set of most likely predecessors $\psi_t(j)$ back to the host (Fig. 4.1).

The recogniser itself consists of two parts: the observation probability computation block, and the Viterbi decoder block (Fig. 4.2).

4.2 Data representation

4.2.1 General

Performing the calculations for speech recognition in the log domain makes them more amenable to implementing in hardware.

In addition, since taking the log of a probability always yields a negative number for any base greater than 1, the log probability is multiplied by a negative constant K, so that all the values encountered are positive. This results in a probability of 1 being represented as 0, and 0 as $+\infty$.

For Viterbi decoding, the dynamic range of the data is sufficiently small (specifically, the values are all probabilities, so correspond to the range $[0,1]$ in the linear domain) to allow them to be represented as fixed-point values. Pruning of unlikely paths through the trellis, which may be performed explicitly in software, happens implicitly in hardware by "removing" values which have exceeded their specified bit width.

In practice, a value which has overflowed in this way can be set to $2^n - 1$, where n is the bit width, which is the equivalent of setting all of its bits to 1.

The same is not true for the observation probability computation. Its inputs are not probabilities, while its outputs are — but often very small ones. Even in the log domain, the dynamic range required means that fixed-point cannot feasibly be used, and floating-point must be implemented instead.

4.2.2 HTK

HTK processes discrete HMMs using 16-bit data. If a probability A is converted to the negative log domain by computing $-\ln(A)$, a 16-bit log-domain integer value would represent the probabilities from 3×10^{-28462} to 1, a range which is far too broad for the purposes used here. A more reasonable range is 10^{-12} to 1, which can be achieved by

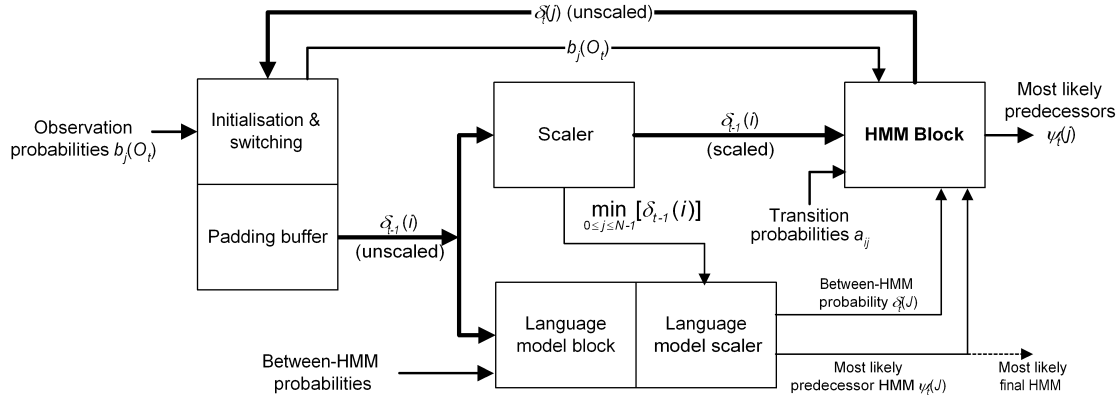

Figure 4.3: Viterbi decoder core

computing $K \ln(A)$, where K equals -2371.8. This is the approach used by HTK.

It was found that for the more complex speech models, 16-bit values are not sufficient to maintain accuracy, and so 24-bit values are used instead, resulting in the range of probabilities being 10^{-3072} to 1. The value of K is kept constant in order to maintain compatibility between implementations.

4.3 Viterbi decoder

The Viterbi decoder consists of six parts, as shown in Fig. 4.3. It takes as its input the observation probabilities $b_j(\mathbf{O}_t)$, be they loaded from a table for discrete HMMs, or calculated for continuous HMMs, and produces each state's most likely predecessor $\psi_t(j)$, which is sent to the PC and processed in software, in order to find the most likely state sequence.

4.3.1 Initialisation and switching

When a new speech file is processed, it is necessary to initialise $\delta_0(j)$ in accordance with equation (2.10). Because of the assumption that, in this case, each HMM will start in its

entry state, $\delta_0(j)$ for each entry state is set to $b_j(\mathbf{O}_0)$, and to infinity for the rest. The initialisation and switching block performs this operation at the start of a speech file.

4.3.2 Scaler

The scaler scales the probabilities, to avoid values exceeding the number of bits used to represent them, as would happen after successive additions.

Firstly, the smallest value is found (corresponding to the *most* likely path), and subtracted from all the values. This does not affect the result of the recognition system, as we are only interested in the paths' *relative* values, not their absolute values.

Then, those scaled values which have exceeded the specified bit width are pruned. This can be effected without using a comparator, by extending the bit width in the HMM block by two in order to compensate for the two additions within that block. Any value whose two most significant bits (after scaling) are not both zero is then considered to have overflowed, and can be pruned. As mentioned in section 4.2, this is done by setting all their bits to 1.

4.3.3 Language model

None of the implementations makes use of a language model. Nonetheless, a language model block is still required, though its purpose is reduced to finding the single most likely predecessor HMM for *all* HMMs. This implements equation (2.15), and scales the result using the smallest value produced by the scaler. The index and probability of the most likely previous HMM is then sent to the HMM block.

In early designs, the language model block came after the scaler in the pipeline, based on the premise that its output must be scaled, and so must first wait for the scaler to finish its operation.

However, it was noticed that the two blocks are independent of each other until they

have finished their respective processing, at which point the smallest $\delta_{t-1}(i)$ value from the scaler is used to scale the output from the language model block.

Given that neither block produces any output until data from all the HMMs have been processed, putting the two blocks in parallel shortens the pipeline, hence reducing the processing time for the Viterbi decoder.

At the end of the observation sequence, we need to know the most likely final state. We assume that the sequence ends with an HMM's exit state, and so this state is simply the most likely predecessor HMM at time T.

A design for a full language model block is proposed in section 4.7 below. If it were implemented, it would occupy the same place in the system as the current block.

4.3.4 HMM block

The HMM processor contains nodes (Fig. 4.4) for implementing equations (2.11) and (2.12). Each one performs these calculations for a single state, taking in $\delta_{t-1}(i)$ from the scaler, the transition probabilities a_{ij} from their store, the observation probabilities $b_j(\mathbf{O}_t)$ — either loaded from a table for discrete HMMs, or calculated in the case of continuous HMMs — and for entry states, the most likely previous HMM and its probability. Accordingly, each node outputs $\delta_t(j)$, sent to the padding buffer, and the most likely predecessor state $\psi_t(j)$, sent to the PC.

As every node depends only on data produced by nodes in the previous time frame (i.e. at time $t-1$), and not the current one, we can — in theory — implement as many nodes as we like in parallel. That, as well as the fact that each node can be pipelined and perform a number of additions in parallel, are what originally made speech recognition a prime target for implementation in hardware. As we shall see in chapter 5, however, other factors place a severe limit on the degree of parallelism we can employ.

That notwithstanding, it is convenient to implement in parallel as many nodes as there

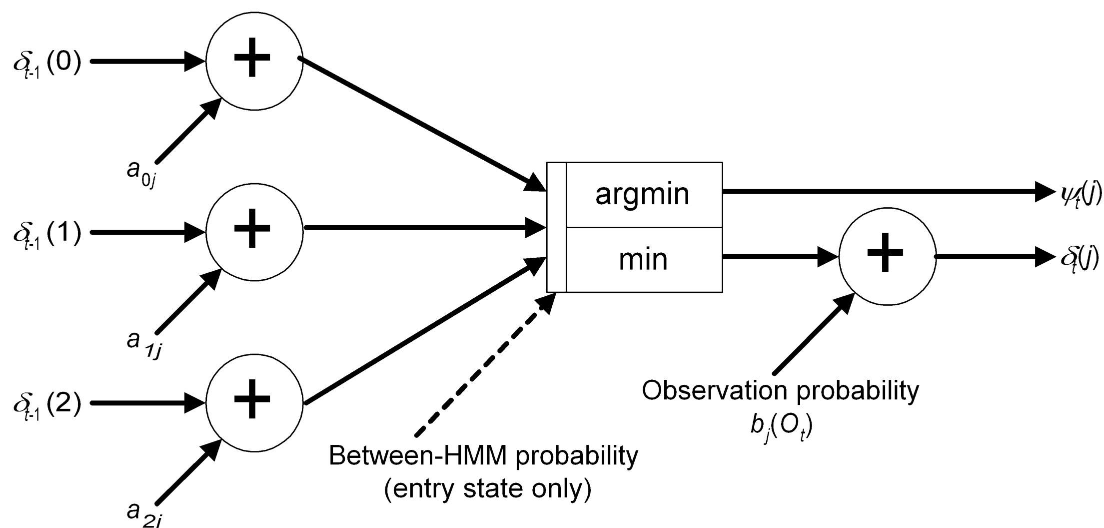

Figure 4.4: Node structure

are states in an HMM, as each one uses and produces the same set of $\delta_t(j)$ values, corresponding to the transitions within one HMM. As a result, the whole data pipeline in the Viterbi core consists of three parallel streams, corresponding to the three states in each HMM.

Since all HMMs use the same nodes, the number of parallel nodes must be greater than or equal to the number of states in each HMM. Three states are used here, since the models used for the implementations all contain that number; but the design could easily be modified if larger HMMs were required.

It is normal practice in speech models to only permit transitions from one state to itself or a later state (left-right or non-ergodic model); but this restriction may not apply to the HMM representing silence, and so for maximum flexibility, the hardware is not constrained in this way.

4.3.5 Padding buffer

Where the total pipeline depth of this circular pipeline is greater than the number of data items stored in it at any one time, then no additional storage is required; indeed, it is likely that processing cycles will be wasted, as is the case for the simpler implementations.

However, if we are attempting to process serially more data items than the pipeline can hold, we must insert a buffer to deal with the extra data.

This is particularly important at startup, when $b_j(\mathbf{O}_0)$ enters the pipeline and becomes $\delta_0(j)$, and must then be stored while we await $b_j(\mathbf{O}_1)$ so that processing can continue.

4.4 Observation probability computation

The observation probability computation block evaluates observation probabilities based on Gaussian mixtures. It processes multiple speech files in parallel, with a Gaussian mixture component (GMC) block dedicated to each one, and the mixture summation block and Viterbi decoder block shared between them.

4.4.1 Parallelism

Whether we are processing one speech file or many, the same model data are used throughout; what differs are the observation data. If, as is the case for these implementations, we only have sufficient bandwidth to read in one file's observation data at a time, we can handle multiple files by reading in the observations successively, interleaving the files, whilst delaying the model data before it enters the corresponding GMC block. This is illustrated in Fig. 4.5. Clearly, each GMC block only accepts its corresponding observation values, ignoring other files' data.

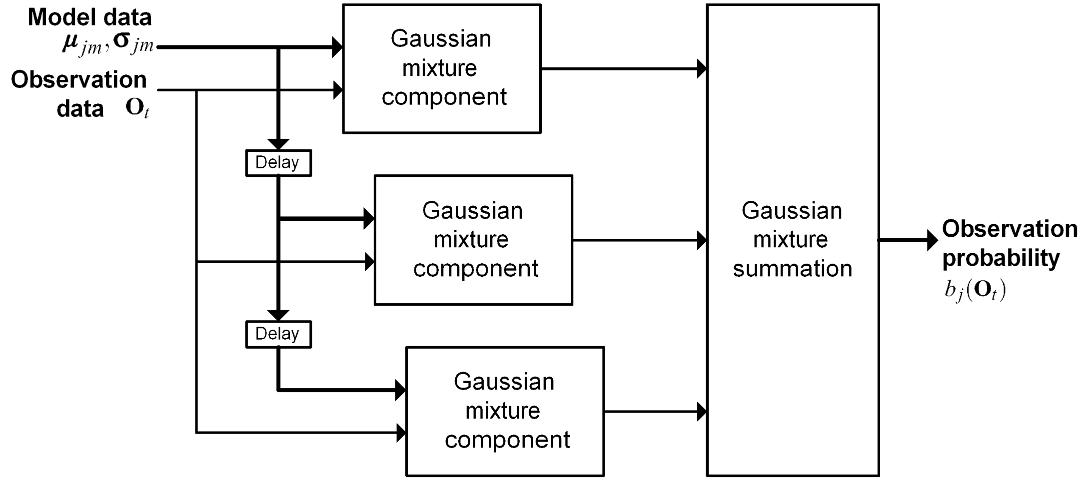

Figure 4.5: Observation probability computation block. The Gaussian mixture summation block is replaced by a multiplexor when the model employs just one mixture component

4.4.2 Gaussian mixture components

The GMC blocks implement equation (2.20). Each block contains a pipeline (Fig. 4.6), with the observations O_{tl}, means μ_{jml} and variance constants V_{jml} as its inputs. It performs a floating-point subtraction, square and multiplication for each element, before summing them, and converting them to fixed-point. Because each state reuses the same observation data at any particular time frame, it is only necessary to read in this data once. The values are stored in the observation buffer, and thereafter output repeatedly until all the states have been processed. It is this saving on bandwidth that enables multiple observation files to be interleaved, since each observation vector is read from RAM just once and used multiple times.

If we had sufficient resources and I/O pins, all of the elements of the observation, mean and variance vectors could be read in at the same time, processed in parallel, and the results summed using perhaps a binary tree adder, so that one observation probability could be produced per clock cycle.

Regrettably, for the most complex design implemented (Cont3_4; see section 5.6), this

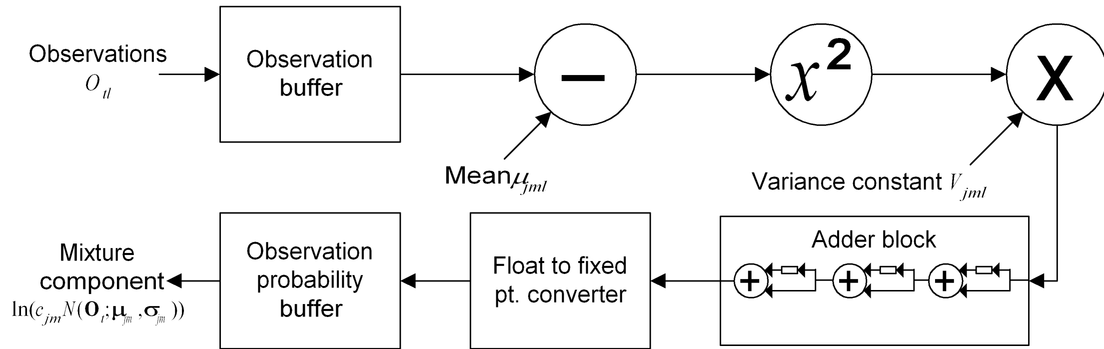

Figure 4.6: Gaussian mixture component (GMC) block. The observation probability buffer is moved to the Gaussian mixture summation block in implementations which feature Gaussian mixtures

would require 10,368 data lines between the processing core and memory! (3 observation vectors $\times$ 4 mixture components $\times$ 27 elements $\times$ 32 bits). For on-chip block or distributed RAM, this could eventually be feasible; for a separate RAM chip, an FPGA or ASIC with 10,000 pins seems a long way off.

In addition, it would require 108 GMC blocks, when just three already fill the FPGA (XCV2000E) used for that implementation. Because of this restriction on resources, three GMC blocks are used for all of the implementations which process multiple speech files in parallel.

The data values for just one element of each mixture component are loaded, and then computed, per clock cycle. The adder block is therefore required to act like an accumulator, though because the latency of a floating-point adder is greater than the one cycle required by an accumulator, it actually consists of a chain of adders (5 where 27 elements are used, 6 for 39 elements).

The constant S_{jm} that appears in equation (2.20) is treated as an additional element, and is included in the summation by assigning it to the variance constant, with the observation value set to 1, and the mean to 0.

Fully parallel and fully serial are opposite extremes for element processing, but of

course, they are not the only options. Resources and bandwidth permitting, the elements could be dealt with as a serial stream, but with a few at a time processed in parallel, in the same way that the HMMs are processed serially, but actually contain three states processed in parallel.

As described in section 4.2, all the log probabilities are multiplied by a negative constant K prior to being used by the Viterbi decoder. This constant is incorporated into the system by pre-multiplying the constants V_{jml} and S_{jm} by K, so that no additional computation is required at run-time.

The floating- to fixed-point converter therefore just shifts the mantissa of the component value as necessary. Very small probabilities, i.e. those that cannot be represented using the bit width chosen for the fixed-point representation, are reduced to zero ($+\infty$ in the negative log domain).

4.4.3 Data storage

Because a result is only produced once every $L+1$ clock cycles (where L is the number of elements in the various vectors), and in order to make it easier to share the decoder and Gaussian mixture blocks between multiple speech files, the mixture component values are stored in a buffer until all of them have been computed, so that they can be sent to the decoder on successive clock cycles.

For the earlier implementations which do not use Gaussian mixtures, a multiplexor is used instead of the summation block, and the buffers sit inbetween the GMC blocks and the multiplexor.

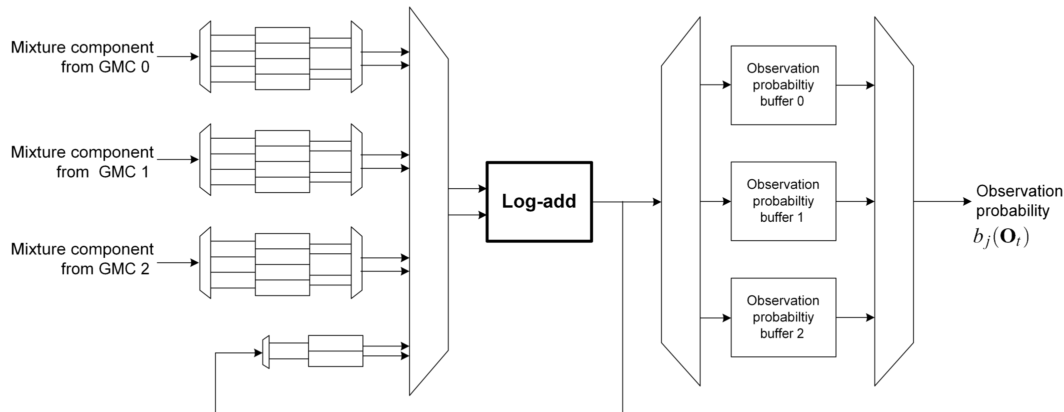

Figure 4.7: Gaussian mixture summation block

4.5 Gaussian mixture summation [35]

The Gaussian mixture summation block implements equation (2.17), using the log-add algorithm described in section 2.7. This novel implementation is more efficient in terms of both resources and clock cycles when compared to the alternatives described in section 3.7, and is well-suited to an FPGA.

4.5.1 Top-level structure

Even with three GMC blocks processing mixture components in parallel, the number of clock cycles between successive values is high enough to allow just one Gaussian mixture summation block to be implemented, and shared between and within mixtures. For the latter, having four mixture components requires three log-add operations (first pair of components, second pair, and the result of those two operations), and these are computed serially. Fig. 4.7 shows the buffers and multiplexors required to achieve this.

For each GMC block, corresponding to one speech file, the mixture components are stored in buffers until all four have arrived. The first two pass through the log-add block, and the result is stored in one of the two registers in the bottom left of the diagram. This

is repeated for the second pair of mixture components. The pair of results from these two operations are themselves then passed through the log-add block, and the result stored in the observation probability buffer corresponding to the GMC block from which the data came. The next GMC block then has its mixture components undergo the same process, and so on, until all of the observation probabilities for all three speech files have been computed.

4.5.2 Data analysis & design

As a hardware implementation, this algorithm seems ideal, since it relies on functions easily realisable on a chip. But in order to give it a significant advantage over the alternative methods described in section 3.7, the look-up table needs to be kept as small as possible without adversely affecting accuracy.

Hence the first step of the implementation is to analyse the data to be used in the table. Software was written to perform the calculations directly, and produce a full set of values. These were then inspected in order to identify patterns which could be used to produce a more efficient design. This involved keeping to a minimum both the number of entries in the look-up table, and the amount of additional logic required.

Inspection of this data reveals that when $K\ln(B/A)$ is 0, $K\ln(1+B/A)$ is -1644. Since all of the outputs are negative, we ignore the sign at this stage. So taking the outputs as positive numbers, as the input value increases, the output decreases, initially at the rate of 1 for every 2 increments of the input, and then more and more slowly. The first consequence of this was that we could ignore the least significant bit of the input, as it did not affect the output by more than ± 1. The other was that for all values of the input above 16,384, the output changed only twice, decreasing from 2 to 1 at 17,471, and then to 0 at 20,077.

The result of this was that a table 8,192 entries deep and 11 bits wide (a total of 11 Kb)

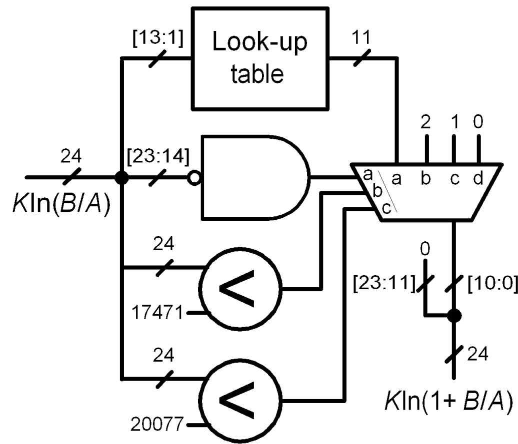

Figure 4.8: Log-add table structure. The ‘and’ gate with negated inputs is equivalent to a comparator checking if the input is less than 16,384, but requires fewer resources. The multiplexor outputs zero (option ‘d’) if the input is greater than or equal to 20,077

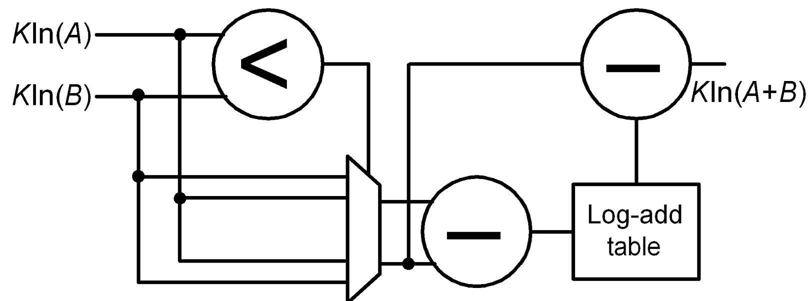

Figure 4.9: Log-add structure

was sufficient to represent all values of the input from 0 to 16,384 (discarding the least significant bit), with the two values above this handled using a couple of comparators, as shown in Fig. 4.8. The only other processing required was a comparator for the two inputs $K\ln(A)$ and $K\ln(B)$, a subtractor to compute their difference, and another subtractor to subtract the smaller (i.e. more negative) input from the output of the look-up table (which is equivalent to adding the smaller input to the negative of the value from the look-up table, required because the numbers stored in the look-up table are positive, the minus sign having been discarded). The architecture of the log-add block is shown in Fig. 4.9.

As a comparison, if we were to convert between domains instead of using this algorithm, there would be the issue of data representation to consider. Using the scaling factor K, a 16-bit value in the log domain translates to a number between 10^{-12} and 1, which would need to be represented as a 40-bit fixed-point value if complete accuracy were required, or floating-point otherwise. With 24 bits, the range is 10^{-3072} to 1, which cannot be sensibly represented in fixed-point. By avoiding a domain conversion, these issues can be circumvented without loss of accuracy.

4.6 Duration modelling

4.6.1 Parallel architecture

Duration modelling provides an opportunity for hardware to significantly outperform software, by taking advantage of the parallelism inherent in this extension to the Viterbi decoding algorithm.

For each duration τ, corresponding to the number of times in which we stay in the same state up to a maximum duration D, we can instantiate a processing element (PE) to compute $\min_i[\nu_t(i,j,\tau)]$, as described in equation (2.24). A proposed architecture is shown in Fig. 4.10.

Each PE takes as its inputs the current observation probability $b_j(\mathbf{O}_t)$, the smallest probability from the scaler $\min_i \delta_{t-1}(i)$, the duration probability $d_j(\tau)$ for current state j and duration τ, and crucially, the *previous* output from the previous stage $\xi_{t-1}(j,\tau-1)$ (defined in equations (2.25) and (2.26)).

This last value must be stored for each stage, before being passed on to the next one. The value sent to the stage corresponding to $\tau = 1$ is computed by the adders and comparator, as shown in the diagram.

The other value produced by each stage, $\min_i[\nu_t(i,j,\tau)]$, is sent to a comparator, ide-

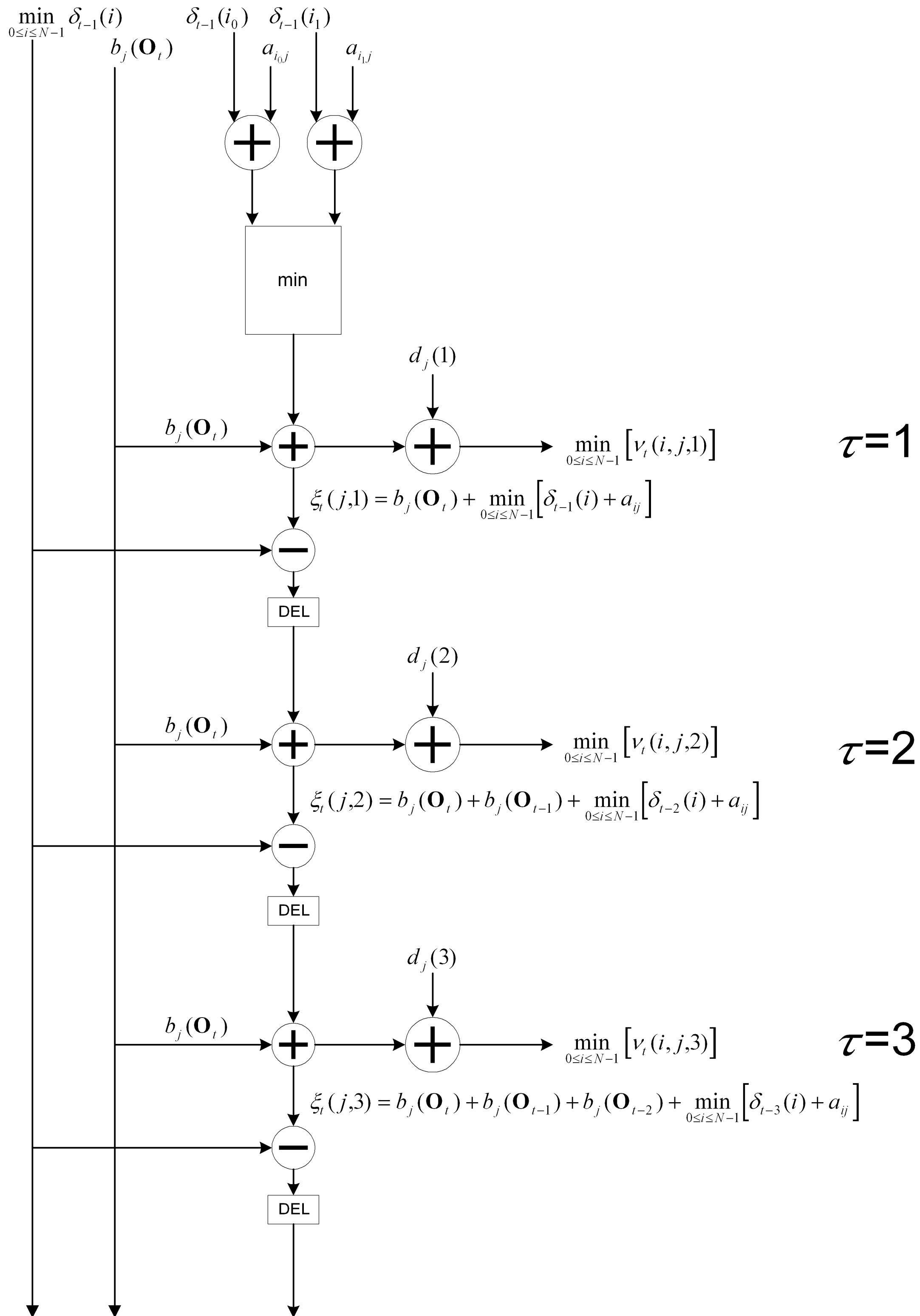

Figure 4.10: Proposed architecture for explicit duration modelling

ally a binary tree comparator, which finds the minimum across all stages, producing $\delta_t(j)$ and $\psi_t(j)$.

The need for simple PEs with local storage makes this part of the system ideal for a SIMD array, and such a platform was indeed used for training and recognition in [37].

4.6.2 Serial architecture

If a single chip is to be targeted, the bandwidth and data storage required for a parallel design could exceed that available. The alternative is a serial design, where a single PE is implemented, the current and previous data are read for one value of τ at a time, and the parallel comparator is replaced by a serial one.

With just one PE, a single $\delta_t(j)$ value will be produced every D cycles, slowing down the rest of the Viterbi decoder in the process. Although each $\delta_t(j)$ is dependent on several other values from previous time frames, the complete set of $\delta_t(j)$ values for the current time t would be stored between observations, meaning that all required data would be immediately available when computing a new $\delta_t(j)$, without any further delays being incurred.

The time taken to produce the observation probabilities *may* be sufficient to offset the delay produced by a serial architecture, though this would depend on the parameters involved, chiefly the number of elements in the observation feature vectors, and the number of mixture components.

4.7 Full language model

If we were to use a full language model, a probability a_{IJ} would be associated with every transition from one HMM's exit state I to another's entry state J. In every time frame, we would then need to find every HMM's most likely predecessor HMM $\psi_t(J)$ by calculating

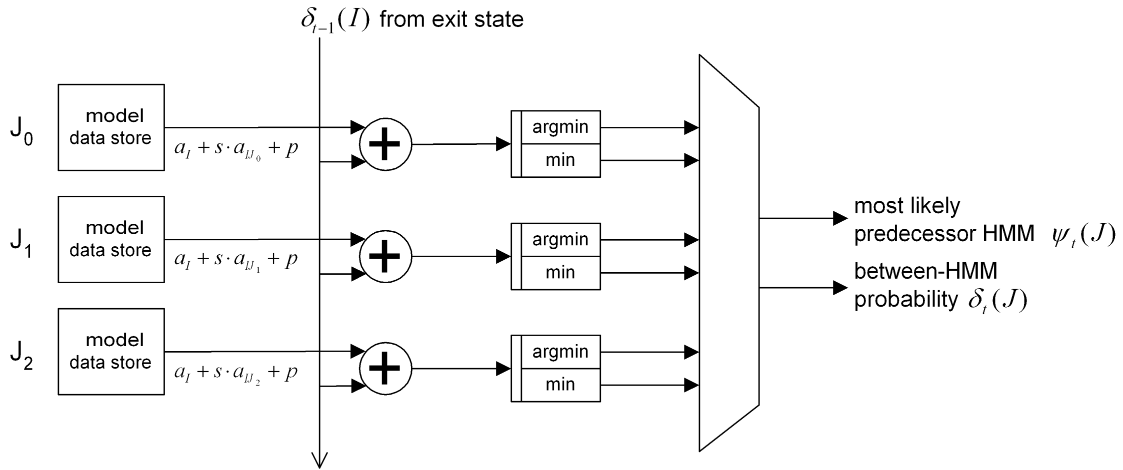

Figure 4.11: Proposed architecture for language model block

its corresponding probability $\delta_t(J)$, as described by equation (2.14).

A fully parallel architecture is shown in Fig. 4.11. In this, the probability $\delta_{t-1}(I)$ associated with an HMM's exit state is added to every between-HMM transition probability (equal to $a_I + s \cdot a_{IJ} + p$) in parallel. A min/arg min block (i.e. a block which finds the minimum of the set of values which pass through it on successive clock cycles, and that minimum's index) finds the most likely predecessor HMM in each case, once all the $\delta_{t-1}(I)$ values have passed through, and a multiplexor then sends each one in turn to the HMM block.

At first glance, it may appear that, as each HMM's data values are computed at the same time, but used on different clock cycles, the input data could be staggered and some hardware reused. However, this is not the case, since the min blocks only produce a value every H cycles (where H is the total number of HMMs), once all $\delta_{t-1}(I)$ values have been input.

The language model block and scaler both take as many cycles to perform their processing as there are HMMs, plus a few additional cycles for arithmetic, etc. Hence, irrespective of the number of HMMs, they can be implemented in parallel, as shown in

Fig. 4.3.

This design requires either storage space for all of the between-HMM transition probabilities, or sufficient bandwidth to be able to load all required values onto the chip at once. It also needs one adder and one min block for every HMM. Again, the requirement for local data storage and high total bandwidth suggests that a SIMD architecture may be more appropriate than a single monolithic chip.

Furthermore, the language model block is not a good candidate for a serial implementation, as rather than requiring around H cycles to do its work, it would take H^2. This is because the min block takes H cycles, and would have to do so for each HMM in succession.

4.8 Further design issues

4.8.1 Control

Control of the system is distributed, and takes the form of HMM identification numbers (and, within the observation probability computation block, element numbers) which travel through the pipeline alongside the probabilities associated with that HMM, with additional identifiers used to signal the beginning and end of a data sequence.

This data stationary control [50] has a number of advantages. During development, it obviates the need for a central control module, making it easier to add blocks into the pipeline without affecting the rest of the system. During simulation and testing, it is clear to see which data is associated with each HMM, and whether additional registers are required to synchronise the different parts of the system.

During the operation of the recogniser, this approach becomes even more useful. There are two independent data streams — the observation probabilities stream $b_j(\mathbf{O}_t)$ and the $\delta_t(j)$ stream — and these streams converge in the HMM block, where they must

be synchronised so that they represent the same HMM (i.e. the value of j is the same for both streams). This synchronisation is achieved by a local control block which, in the event that one stream is ahead of the other, suspends the subsystem generating that stream until they are synchronised once again.

In addition to the various control signals, numerous shift registers are employed to synchronise the data and control streams. For clarity, these, and the control signals, are not included in the diagrams in this chapter.

4.8.2 Pruning

As mentioned in sections 4.2.1 and 4.3.2, the least likely paths through the trellis are implicitly pruned when their $\delta_t(j)$ values exceed the allowed bit width.

While there can be performance advantages in removing pruned HMMs from the data stream, there are consequences to having the pipeline contain a data stream whose length can change on each trip around the system.

The design assumes a continuous stream of HMM data, bookended with start and end control signals. If an HMM were to be removed from the stream, the next block in the pipeline (presumably the HMM computation block, if the scaler were doing the pruning) would need to be suspended for a clock cycle to compensate.

If the HMM block were then to re-insert a pruned HMM *in situ*, the previous block would have to be suspended for a cycle while this took place. HMMs could instead be re-inserted at the end of the stream without penalty, provided that when processing multiple speech files, there is a sufficient gap between the end of one and the start of the next.

If necessary, these problems could be rectified by moving the padding buffer to between the Scaler and HMM block. It should be borne in mind, however, that for the later implementations, the time taken for the Viterbi decoder to do its processing was much smaller than that of the observation probability block, and even without any pruning, the

Viterbi core stood idle most of the time.

The end result is that it is far simpler to have a data stream whose size does not change, and hence have a pipeline which does not have to be continually suspended in places, than to add and remove data items. Pruned items can instead be marked with a probability representing impossibility.

4.8.3 Non-emitting states

HMMs are sometimes represented as having non-emitting or null states, typically with one as an HMM's sole entry state, and another as its exit state.

In the case of the HTK models used here, each HMM has a single non-emitting entry state, with a single transition of probability 1 to the first emitting state. Similarly, the last emitting state has a transition to a non-emitting exit state, with the associated probability a_I being the probability of a transition from that HMM to any other, as introduced in section 2.4.

Non-emitting states, while perhaps adding convenient "packaging" to an HMM, do not aid its processing. In these designs, the entry states do not provide any additional information, and so are removed, with the first emitting state of each HMM now becoming the entry state.

For the exit states, a_I is subsumed into the language model, making the state redundant, and so the last emitting state becomes the exit state. If an HMM were to have more than one exit state, this too could be incorporated into the language model.

4.9 Summary

This chapter has set out the designs for implementing various parts of the speech recognition algorithm in hardware. Starting with comments as to how data may be represented

in such a system, the Viterbi decoder and its component parts have been described. This has been followed by the observation probability computation blocks, and the Gaussian mixture summation block.

Designs have been proposed for duration modelling and the language model block, and other general issues related to the design have been covered.

Of particular noteworthiness in these designs are the ability of the observation probability computation block to process multiple speakers in parallel, despite bandwidth limitations, and the small and efficient hardware design for the log-add algorithm.

Having proposed these designs on paper, they must then be realised in silicon; the FPGA implementations are described next.

The presence of humans, in a system containing high-speed electronic computers and high-speed, accurate communications, is quite inhibiting.

Stuart Luman Seaton (1906–)

5 Implementation

This chapter deals with the application of the designs to hardware, and the software that accompanies it. Whereas, up to this point in the thesis, it has been important to separate the theory and designs from any particular implementation, it is now necessary to describe how these designs were turned into a real, operational, speech recognition system.

Described below are details of the hardware and software used, followed by each of the implementations. Detailed timing, resource usage and accuracy results are given in chapter 7.

5.1 System environment

5.1.1 Hardware

The FPGAs used are two Xilinx [71] Virtex-series devices. The earlier designs are implemented on the Virtex XCV1000 BG560-6, with the later, more complex designs using the larger Virtex-E XCV2000E BG560-6. In all cases, the code for the designs is written in

VHDL.

In each case, the FPGA is situated on a Celoxica [63] RC1000-PP development board. The RC1000 is a PCI card, housed inside a PC, which features a number of resources and data paths accessible by both the FPGA and the host PC. Chief among them is 8 Mb of RAM, organised as 4 banks of 2 Mb, each with a 32-bit data bus to the FPGA and PC; arbitration logic is provided to allow ownership of each bank to be transferred between the FPGA and PC. In addition, there are two 1-bit general-purpose data lines, one in each direction, and two 8-bit data registers, also unidirectional and with handshaking built in.

An interface/wrapper was written for the RC1000, with the idea that any user core could be inserted into it, without the user having to know the details of how the FPGA communicates with the outside world — essentially the hardware equivalent of an API[1].

The host PC is a Pentium-III 450, which houses the two RC1000 cards.

5.1.2 Software

The software component, called simply "Speech", is written in C++ under Microsoft Visual Studio. It performs numerous functions related to the speech recognition system, including:

- Performing the same recognition tasks as the hardware, using equivalent algorithms (unless it would be particularly inefficient to do so);
- Outputting the best sequence of recognition units, with time indices, in the same format as used by HTK, and irrespective of which platform is used for the recognition;
- Interfacing with the RC1000 board when recognition is being performed in hardware;

[1] As a result of the interface being posted on Celoxica's website, more correspondence has been received about this than about the application for which it was originally written!

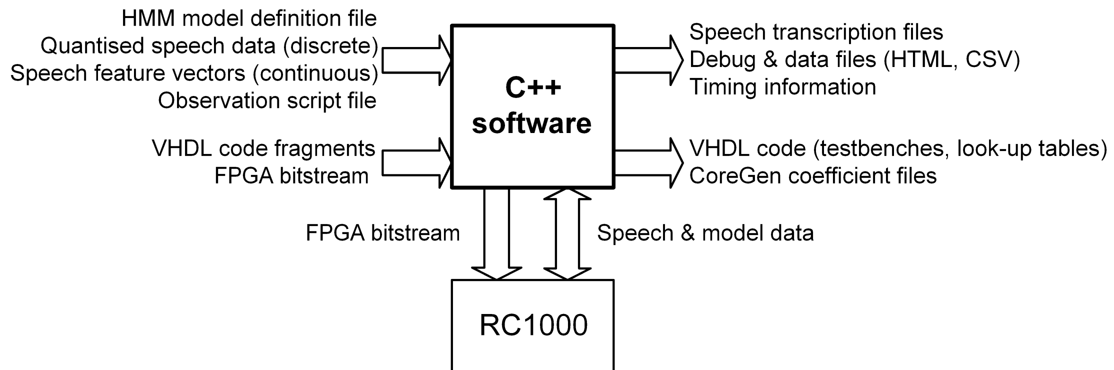

Figure 5.1: Software input and outputs

- Generating timing data, printed to the screen during operation;
- Generating debug data, output as an HTML file, with the data laid out in such a way as to make comparison with the VHDL simulator as simple as possible;
- Generating VHDL code and data files for implementing look-up tables, etc.;
- Generating VHDL testbench code for simulation.

Because of the complexity of the hardware design, having a toolkit capable of all of these functions is essential.

The data flow into and out of the software is shown in Fig. 5.1.

The software consists of a number of principal classes, each instantiated once, and connected as shown in Fig. 5.2. The classes are arranged in three informal layers: the top-level layer, containing the executive class (**Speech**); the interface layer, which contains the classes whose function is to communicate with the outside world, specifically to and from data files and the user (**IOHandler**), the RC1000 (**RC1000Handler**), VHDL files (**VhdlWriter**), and the Windows registry (**RegHandler**); and the kernel layer, where the actual data processing takes place, including the model data processing prior to recognition in hardware or software (**HmmDefs**), and the class which performs the recognition

in software (**Column**), including Gaussian mixture summation (**LogAdd**). These are described in more detail below.

The software does not require any advance knowledge of the type or size of the model being processed. Rather, it extracts all relevant information from the HMM definition file and recorded speech files, and allocates memory dynamically at run-time. The same software is used for all implementations.

Speech

This is the top-level class, containing the "executive" function which processes the command-line parameters and calls functions in the other principal classes. It also performs backtracking once the decoding has been performed in hardware or software.

Although backtracking could be considered as data processing, and so might be better placed in the kernel layer, **Speech** is the only part of the software that bridges both the hardware and software implementations of the recogniser. The backtracking process is the same irrespective of the platform on which the recognition is done, hence its inclusion here.

IOHandler

This block deals with most of the file I/O, as well as printing information to the screen. Its inputs are hmmdefs, the HMM definition file created by HTK, which is passed to **HmmDefs**, along with the speech observation files (*.mfc), and the script files (*.scr) containing the list of speech files to be processed.

Once processing is complete, it outputs various data files, including the recognition transcription files (*.mlf), and debugging information in HTML and CSV (spreadsheet) format.

IOHandler also stores data relating to the observations, but not the model, namely

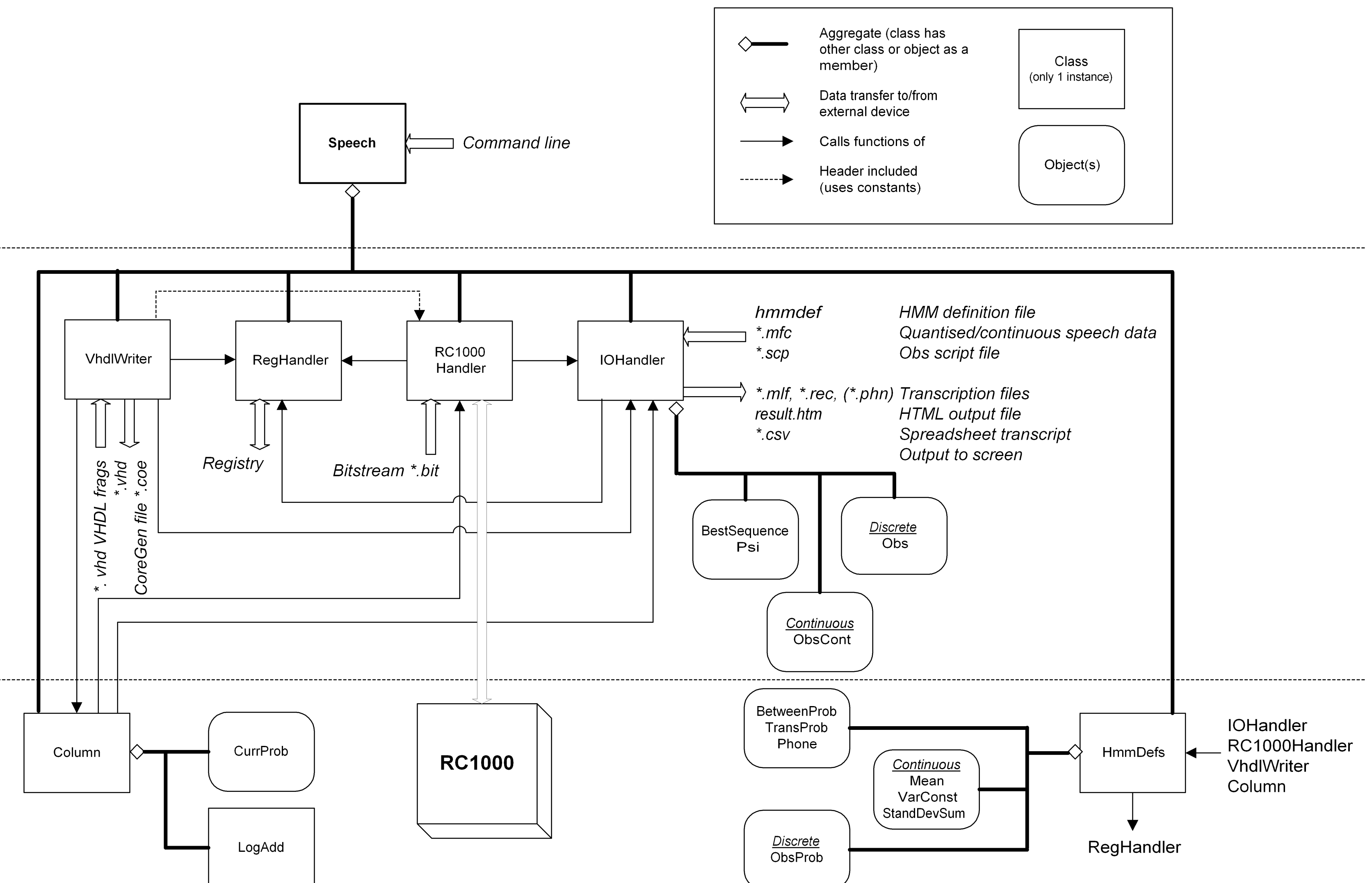

Figure 5.2: Software object model

the observations themselves (Obs or ObsCont for discrete or continuous, respectively), the most likely predecessors (Psi), and the resulting most likely sequence of utterances (BestSequence).

RC1000Handler

This provides the interface between the software and the RC1000. It reads in the bitstream file and uses it to configure the FPGA. It is also responsible for all the data transfer between the host PC and the RC1000's RAM, which encompasses writing the model and speech data, and reading back the most likely predecessor information.

VhdlWriter

In order to facilitate testing, VHDL testbench files are generated by this class, based on the same speech and model data used in the rest of the system.

In addition, this block generates look-up table data for VHDL and Core Generator blocks, as well as code for the binary-tree comparators used in the Viterbi decoder core, which would be particularly difficult to parameterise just using `generate` statements.

RegHandler

Configuration details are stored in the Windows registry, including file and directory names, and model parameters (such as grammar scale factor s, word insertion penalty p and maximum duration D). This class provides functions for accessing and modifying this information.

HmmDefs

This block processes, at the initialisation stage, the HMM model data as provided by HTK. For all models, it extracts the transition probabilities a_{ij} (TransProb), and the list

of monophones or biphones/triphones (Phone), as well as computing the between-HMM probabilities based on equation (2.15).

For the discrete HMM model, it extracts the codebook entries for the observation probability $b_j(\mathbf{O}_t)$ (ObsProb). For the continuous HMM models, it extracts and computes the values used in equation (2.20), namely S_{jm} (StandDevSum), V_{jml} (VarConst) and μ_{jml} (Mean).

Once processed, these values are then made available to other classes as necessary. Note that none of these computations form part of the critical path, as they are performed once at startup, with the data stored thereafter.

Column

Column[2] performs Viterbi decoding in software. The current values of $\delta_t(j)$ are stored in CurrProb.

LogAdd

This class generates the look-up table data, used by both hardware and software, for Gaussian mixture summation, as well as performing the summation when recognition is done in software. In addition, it contains code that was used to analyse the table data in order to work out how best to implement it in hardware.

5.1.3 Speech data & models

The speech files used for the testing and training of all implementations are taken from the TIMIT database [70], a collection of speech data designed for the development of speech recognition systems. The groups consist of samples of continuous speech from each of

[2] The name comes from the earliest FPGA implementations, where the hardware HMM block was envisaged as representing a column of the state-time trellis, at the current time t.

Table 5.1: Speech models, showing the number of HMMs, elements, Gaussian mixture components and parallel speech files, and the maximum duration D. Model data format refers to the codebook index for the discrete model, and the observation, mean and variance values for the continuous models

Name	HMMs	El	Mix	Files	D	Model data format	Obs. prob. width (bits)
Disc1	49	(1)	1	1		8-bit fixed-point	15
Cont1	49	39	1	1		32-bit floating-point	24
Cont1P	49	39	1	3		32-bit floating-point	24
Cont3P	634	39	1	3		32-bit floating-point	24
Cont3_4	634	27	4	3		32-bit floating-point	24
Cont3_4D1	634	27	4	3		32-bit floating-point	24
Cont3_4D	634	27	4	3	15	32-bit floating-point	24

8 American English dialect regions, from both male and female speakers. For training, several hundred annotated recordings from the TIMIT training set were used; for testing, 160 recordings from the test set were employed.

The files were pre-processed with HTK. For the discrete implementation, each file consists of a sequence of codebook entries, whereas the continuous implementations use mel-frequency cepstral coefficients (MFCC).

Details of the implementations are summarised in Table 5.1. There are four distinct models used: discrete monophone (Disc1), continuous monophone (Cont1, Cont1P), continuous biphone/triphone (Cont3P), and continuous biphone/triphone with four Gaussian mixtures (Cont3_4, Cont3_4D1, Cont3_4D).

The duration model data values used in Cont3_4D were generated purely from the transition probabilities, such that $d_j(\tau) = a_{jj}^{\tau-1}$, as no suitable values were otherwise available. While this had the effect of reducing the recognition accuracy, rather than increasing it, as a representative model and proof of concept, it was sufficient.

5.1.4 Virtex/Virtex-E FPGAs

The Xilinx [71] Virtex and Virtex-E FPGAs have the same underlying architecture, but differ primarily in terms of quantity of resources, feature size (0.22 μm and 0.18 μm respectively), and maximum clock rate.

The basic building block is the configurable logic block (CLB), which contains two slices. Each slice contains two look-up tables (LUTs), each of which can be configured as any logic function with one output and up to four inputs, a 16-bit ROM, a 16-bit RAM, or a shift register (SRL) of up to 16 stages. Also in the slice is dedicated carry logic which allows CLBs to be connected together to form carry chains for addition, subtraction and comparison operations, as well as being used in multiplier cores. The outputs from the slices can be registered using the two flip-flops (FFs), each of which can act as a register or level-sensitive latch, with clock enable and synchronous or asynchronous set and reset, and be clocked on a rising or falling clock edge (but not both).

Also available are Block RAMs, which are 4,096-bit dual-port memories, whose address and data buses have configurable widths. The Block RAMs are commonly combined in cores to form larger memories of specified dimensions.

The Virtex XCV1000, used for Disc1 and Cont1, contains 12,288 slices (a total of 24,576 LUTs and FFs) and 32 Block RAMs (131,072 bits, or 16 Kb). The Virtex-E XCV2000E, used for the other implementations, contains 19,200 slices (38,400 LUTs and FFs) and 160 Block RAMs (655,360 bits, or 80 Kb).

5.2 Disc1 [31][33]

Discrete, 49 monophones, 256-entry codebook

Implementing speech recognition on a programmable logic device began with the simplest possible system - a monophone model making use of discrete HMMs.

For this implementation, the pre-processed speech observations consisted of quantised

8-bit values, treated as addresses into 256-entry, 15-bit-wide look-up tables as generated by HTK. Each state had such a codebook, which was stored in off-chip RAM. The transition probabilities were stored on-chip in Block RAM, and the small number of between-HMM probabilities were stored in distributed RAM. Internally, scaled probabilities were stored as 16-bit log-domain values, with overflows detected as necessary. The software used the same bit widths.

The design was implemented in three versions, each able to process a different number of HMMs in parallel.

5.2.1 49-HMM implementation

The first version attempted to process all 49 HMMs in parallel, but did not fit into the XCV1000, requiring 100% of the FPGA's slices, which along with a $\delta_t(j)$ data bus over 2,000 bits wide, resulted in a design which could not be routed.

In addition to the resources used by the HMM processing block, the 49-stream binary-tree comparators, as used by the scaler and between-HMM probability block, tied up a significant amount of the chip's arithmetic logic.

Even if it had been possible to route the design, or if a larger chip had been used, any benefit produced by the parallel processing of the HMMs would have been lost due a lack of bandwidth between the FPGA and the outside world.

5.2.2 7-HMM implementation

In order to reduce the required resources, a second version was implemented which cut the number of parallel HMMs to 7, with the other modules in the system reduced in size accordingly. The final design required 70% of the slices, and routed successfully. It ran at a maximum clock frequency of 31 MHz.

The design required 26 cycles to process a single observation, but because the internal

data bus was larger than the off-chip RAM data bus, further delays were incurred: 36 cycles to read in the observation probabilities $b_j(\mathbf{O}_t)$ from RAM and 20 to write the predecessor information $\psi_t(j)$ back to RAM. As some of the RAM accesses could take place while data was being processed, the total was slightly less than the sum of these three values, being 77 cycles.

Experiments showed the average time for a complete observation cycle (taken to be the mean time to read the observation values from RAM, process them, and write the predecessor information back to RAM) to be 2.1 μs.

However, this implementation was still not satisfactory, as there was a significant bottleneck when it came to reading from and writing to off-chip RAM. In addition, any implementations based on a more complex algorithm (e.g. continuous HMMs) would have had no more space on the FPGA with which to perform additional calculations as required.

5.2.3 1-HMM implementation

In order to overcome these problems, a third version was designed which dealt with just one HMM at a time, reducing the resource usage, and matching the internal bandwidth with that available for accessing the RAM, thereby reducing the delay incurred by the second implementation.

This design was successfully implemented, requiring 1,600 slices, equal to 12% of the XCV1000's resources. It operated at 55 MHz, with a 62-stage pipeline. Most of this latency was due to the Scaler and between-HMM probability block requiring the data for all 49 HMMs to pass through them before they could produce a result.

One consequence of processing just one HMM at a time was that the effective delay due to RAM accesses was reduced, as data from RAM could enter the pipeline as soon as it was read, rather than being buffered first as was done in the previous implementation. Hence a complete observation cycle took 1.13 μs.

As the pipeline was circular, all of the HMM data had to pass through it before new data could be processed, but since the pipeline depth was longer than the data length, 11 cycles out of every 62 were wasted.

While this implementation had less of a problem with RAM-FPGA bandwidth than the more parallel versions, the system had to pause from time to time while predecessor state data was written to RAM. This was aided by queuing the data in the FPGA while it was waiting to be written. A consequence of this was that data was written to RAM continually, including during cycles when no new data was being produced.

Taking this into account, the average observation time went up to around 2.0 μs per 10 ms observation — more than 4,900 times real time.

5.3 Cont1 [32][33]

Continuous, 49 monophones, 1 mixture component, 39 elements

A block was added to the discrete HMM system for computing the observation probabilities as defined in equation (2.20), and structured as shown in Fig. 4.6. As before, software was written which was as as functionally similar as possible to the hardware implementation.

Although it would have been simpler to use fixed-point arithmetic for the GMC (Gaussian mixture component) calculations, the dynamic range of the data was such that this was not feasible. Hence the adder, multiplier and squarer were designed for floating-point, based on IEEE 754 [16], though limited to the functionality required for this specific design. For example, the exception handling described in the standard was not implemented, and the squarer ignored the sign bit of its input.

The continuous observation vectors extracted from the speech waveforms, and the mean and variance vectors for each state, consisted of 39 single-precision floating-point values. The term S_{jm} appearing in equation (2.20) was treated as an additional element.

As just one mixture component was used, c_{jm} was set to 1, hence $\ln(c_{jm})$ became 0.

The observation probabilities calculated from this data tended to be one or two orders of magnitude smaller than their discrete HMM counterparts, so it was necessary to increase the width of their fixed-point, log-domain, equivalents from 16 bits to 24. This value was settled on after comparing the results of the software with HTK for different maximum bit widths.

The design occupied 5,590 of the XCV1000's slices, equal to 45%, and was capable of running at 47 MHz, though the speed of the off-chip RAM allowed a maximum clock rate of 44 MHz. (1-cycle reads were used for this implementation, whereas for the discrete one 2-cycles reads were used, allowing a higher clock speed; in both cases, writes to RAM required 2 cycles).

The slowest part of the system was the observation probability computation block (consisting of a GMC block without the additional multiplexors and other logic used in later implementations), which required 6 adders in the adder chain, and which wrote a single value to its buffer every 40 cycles. The contents of the buffer were sent to the decoder only when *all* of the HMMs' probabilities had been computed. This was originally done because the decoder was designed to process its data on consecutive clock cycles, but also had the advantage of laying some of the groundwork for the later parallel implementations.

Consequently, the Viterbi decoder core sat idle for most of the 134 μs which the system took to compute all of the observation probabilities for each observation, and then produce the predecessor information. This did at least remove the bottleneck in writing this information to RAM.

Whereas the mean and variance values for the observation probabilities were stored off-chip, the transition probabilities were stored in Block RAM, and the between-HMM probabilities were stored in distributed RAM.

5.4 Cont1P [34]

Continuous, 49 monophones, 1 mixture component, 39 elements, 3 files in parallel

As described in section 4.4.2, a convenient way of taking advantage of the spare processing time, and the bandwidth freed up by only reading in the observation once, rather than for each state, is to implement more observation probability computation blocks, operating in parallel on different observation data and the same model data.

Now using a Virtex XCV2000E, it was found that there was sufficient space to use three GMC blocks. While this design could be used to process one speech file three times as fast as before, it was felt that in a real-world application, the speech data is more likely to be presented in real time, so three different files are processed at once instead.

As described earlier, the files are read in and stored one after the other, and the model data delayed accordingly for the second and third GMC blocks. The three sets of observation probabilities are stored until all of them have been computed, and are then sent in sequence to the Viterbi core. The three sets of data are interleaved within the Viterbi core, and stored in the padding buffer and within the pipeline while the next set of observation probabilities are being calculated.

The full design occupies 72% of the XCV2000E's slices, requiring 19,000 LUTs, 15,000 FFs, and 30 Block RAMs, with each of the three GMC block using around 4,400 LUTs and 3,700 FFs. The Viterbi decoder is somewhat smaller, using 1,600 LUTs and 2,800 FFs. The whole thing runs at 50 MHz.

The average time taken to process one observation is 39.3 μs. As expected, this is three times faster than the previous implementation which only processed one speech file at a time.

5.5 Cont3P [34]

Continuous, 634 biphones/triphones, 1 mixture component, 39 elements, 3 files in parallel

Moving from 49 monophones to 634 biphones and triphones requires a significant increase in storage space, both on and off chip.

With only the minimum of changes made to the original implementation, the initial version of this larger model required 143% of the FPGA's slices. This was dealt with by using spare Block RAMs as delay elements, in place of shift registers. While Block RAMs cannot act as delay elements on their own, with a little external logic in the form of read and write address pointers (making use of the RAMs' dual ports), the same functionality can be realised.

Additional logic is also required to flush the RAMs when reset. As all of these blocks are within the Viterbi decoder core (specifically, the $\delta_t(j)$ padding buffer, and two more in the scaler), there is sufficient time at reset to flush the blocks while the first set of observation probabilities are being processed.

Hence the slice usage was reduced to 77%, including 22,000 LUTs, 18,000 FFs, and 138 Block RAMs (out of 160). For this implementation, each GMC uses 4,500 LUTs and 5,000 FFs. The Viterbi decoder uses 2,500 LUTs and 2,200 FFs.

This implementation runs at 33 MHz. The average time per observation for the hardware is 769 μs, which is 13.0 times real time.

5.6 Cont3_4

Continuous, 634 biphones/triphones, 4 mixture components, 27 elements, 3 files in parallel

Continuing with the incremental development of the algorithm, a system was implemented based on a biphone/triphone model with the same number of HMMs as for Cont3P,

but with 27 elements per vector (reducing the number of adders in each GMC block's adder chain from 6 to 5), and crucially, 4 Gaussian mixture components. This design incorporates the Gaussian mixture summation block described in section 4.5.

As described previously, the time taken for the GMC blocks to process their data means that just one Gaussian mixture summation block is required, with a single decoder block as before.

After further redesign, which saw the Block RAM used as a RAM (observation buffer), ROM (log-add table, transition probabilities), FIFO (observation probabilities), and delay element ($\delta_t(j)$ padding buffer, scaler control data and probability data buffers), this implementation ended up being slightly smaller than its predecessor. It required 70% of the slices, including 19,000 LUTs, 17,000 FFs, and all 160 Block RAMs.

The design runs at 43 MHz, giving a time per observation of 1,652 μs, which is just over 6 times real time.

5.7 Cont3_4D1

Continuous, 634 biphones/triphones, 4 mixture components, 27 elements, hardware/software hybrid

Given the size of Cont3_4, it was clear that a design that added duration modelling to the previous one would not fit on the FPGA that was available. Fortunately, the PC housing this FPGA also contained a second RC1000 development board, with another XCV2000E on it. It therefore made sense to partition the design between the two boards.

With bandwidth continuing to be a crucial issue, data transfer between the FPGAs can be minimised by placing the observation probability block in the first FPGA, with its results being sent to the second FPGA, where the Viterbi decoding, now incorporating duration modelling, would be performed.

Once the previous design had been reduced to an observation probability processor,

it was tested within what had become a hybrid system, with the observation probabilities computed in hardware, and the Viterbi decoding performed in software.

The implementation occupied 52% of the FPGA, using 14,000 LUTs and a similar number of FFs, with 25 Block RAMs. It operated at 62 MHz. With the software running in debug mode on the host PC, the time per observation is 17,529 μs, more than 10 times slower than Cont3_4.

5.8 Cont3_4D

Continuous, 634 biphones/triphones, 4 mixture components, 27 elements, duration modelling, 2 FPGAs

Work was started on a serial duration modelling implementation, based on that described in section 4.6.2.

It soon became apparent, however, that this was not an effective use of such a large FPGA. A lack of bandwidth limited the implementation to a single PE, which would have taken up only a small part of the chip. As suggested earlier, this application is better suited to a SIMD architecture with each PE having its own local memory, and not an FPGA.

Cont3_4's observation probability blocks produce 3 values every 112 clock cycles (27 elements per vector plus the additional constant element $\times$ 4 mixture components), or 1 every 37 cycles. A serial duration modelling block would produce 1 value every D cycles, so as long as D is less than 37 (in this case), the Viterbi decoder should be able to keep up.

5.9 Summary

This chapter has described the various implementations which are key to this research. Building on the theory and designs covered in earlier chapters, the system has been im-

plemented in software and in hardware.

Having described the PC and FPGAs on which the system was implemented, along with the speech models themselves, and the structure of the software, the implementations have been presented. These are summarised in Table 5.2, which besides providing a brief description of each model, lists the number of HMMs, codebook entries (Disc1 only), vector elements, mixture components, files processed in parallel, and maximum duration (Cont3_4D only).

An analysis of the requirements of implementing speech recognition in hardware and software is presented next. The results produced by the implementations follow that, and draw on the analysis to verify the predictions made.

Table 5.2: Summary of implementations

Name	Type	Recog. unit	HMM	Code-book	El	Mix	Par. files	D	FPGA	Notes
Disc1	discrete	monophones	49	256		1	1		XCV1000	
Cont1	continuous	monophones	49		39	1	1		XCV1000	
Cont1P	continuous	monophones	49		39	1	3		XCV2000E	
Cont3P	continuous	bi/triphones	634		39	1	3		XCV2000E	
Cont3_4	continuous	bi/triphones	634		27	4	3		XCV2000E	
Cont3_4D1	continuous	bi/triphones	634		27	4	3		XCV2000E	$b_j(\mathbf{O}_t)$ in HW, decoding in SW
Cont3_4D	continuous	bi/triphones	634		27	4	3	15		SW only

We are just statistics, born to consume resources.

Horace (65–8 BC)

6 Requirements analysis

In the course of implementing speech recognition in programmable logic, it became apparent that while hardware can typically outperform software, the degree of speedup is dependent on a number of factors, including the size and complexity of the model, the hardware resources available, and the bandwidth (i.e. quantity of data transferred between the chip being used and RAM per unit time).

Analysing this relationship is useful, both to see if the implemented designs behave as the analysis might predict, and for the purpose of looking to the future, in order to describe the resources that may be needed to implement more complex speech systems. It also helps to divorce the design from any particular hardware platform, such an association being an inevitable consequence of implementing a design — however device-independent it may attempt to be — on a real chip.

So presented here is a requirements analysis, independent of any particular hardware. The text of this chapter appears in a modified form in [36].

6.1 Number of HMMs

All of the hardware implementations process just one HMM at a time for a given speech file, so the time taken to output the data for one observation, for both hardware and software, is directly proportional to the number of HMMs. This value has little impact on the hardware resources required, due to the relatively small size of the HMM computation block.

The same is true of the number of elements in the observation feature vectors for continuous HMMs, which are also processed serially.

6.2 Number of HMMs implemented

The decision to process just one HMM at a time is due to limitations on bandwidth, since the FPGA being used does not have enough pins to allow the required data to be loaded onto the chip fast enough to take advantage of the extra parallelism available on-chip. Greater parallelism requires more model data to be read onto the chip, and predecessor data to be written to RAM. Specifically, bandwidth is proportional to the number of HMMs implemented, and is independent of the number of HMMs in the model.

Hence, bandwidth permitting, the time per observation for hardware is inversely proportional to the number of HMMs implemented in parallel (i.e. the number of nodes implemented). Clearly more resources are required if implementing more HMMs; more adders are used, and the comparators in the blocks dealing with scaling and the language model will have more inputs. The internal data bus width also increases.

6.3 Number of mixture components & number implemented

Whereas the small size of the HMM processing core means that bandwidth rather than resources is the limiting factor, for Gaussian mixtures, both affect how many mixture components we can implement in parallel. Three observation probability processing cores can be fitted onto an XCV2000E, and so it can process three speech files simultaneously — but for each file, just one mixture component is computed at a time. Bandwidth is saved because they all use the same model data, requiring the data to be loaded onto the chip just once. For a multiple mixture implementation, each mixture component has its own mean and variance data, so the bandwidth requirement is scaled by the number of components chosen to be implemented in parallel.

Accordingly, the time taken per observation is proportional to the number of mixture components for both hardware and software, and inversely proportional to the number of components implemented in parallel in hardware.

6.4 Language model

In the implementations presented herein, a language model is not used, so the corresponding block in Fig. 4.3 is simpler than it would be if one were. Based on the proposed design for the language model block illustrated in Fig. 4.11, an FPGA with sufficient on-chip RAM could store locally all the data associated with a language model. This would remove any bandwidth overhead, and all the HMMs' most likely predecessor HMMs could be processed in parallel while the scaler is doing its computation. Hence such an implementation would take no longer than the current one.

This implementation requires one adder for each HMM, and storage for H^2 probabilities, H being the total number of HMMs. Alternatively, a serial implementation,

performing just one addition per clock cycle, would load the between-HMM probabilities onto the FPGA one at a time, and would take of the order of H^2 cycles. In software, the delay incurred would also be proportional to the square of the number of HMMs. For both software and serial hardware, the delay for a small model would be small compared to the overall processing time. However, as the number of HMMs increases, this delay would dominate the timing.

6.5 Duration modelling

Explicit duration modelling is one area where hardware has the potential for major speed-up over software, subject to any limitations on bandwidth and resources.

As described in sections 2.8 and 4.6, each stage, representing the number of occasions the system has stayed in the current state, requires two additions. One of the results from each stage is sent to a comparator which computes the most likely duration across all stages. The other result is scaled (requiring a subtraction) and stored, before being passed to the next stage.

In software, we repeat these calculations D times (where D is the maximum duration, typically of the order of 15 to 30). In hardware, we would like to implement all the stages in parallel, with buffers separating the data between stages, as illustrated in Fig. 4.10. However, the storage space required for the buffers and for the duration probabilities easily exceeds that available on current FPGAs. If we store either set of values off-chip, bandwidth restricts how many stages' data can be read or written at one time.

The result is that the delay for the duration modelling part of the system, if partly or wholly serial, is proportional to the number of stages for both hardware and software, and inversely proportional to the number of stages implemented in parallel.

If, however, this were to be implemented on a SIMD array, with one processing ele-

ment per stage, and each processing element having local memory, this could overcome the bandwidth issue. The delay would then be independent of the number of stages, while the bandwidth increase, and the speedup over software, would be proportional to it.

6.6 Parallel files

Processing multiple speech files or speakers in parallel can further increase the speedup of hardware and software, and is possible because of spare bandwidth while reading data on to the chip, and the under-utilisation of the decoder core, resulting in no significant time penalty. Each file implemented in this way requires its own observation probability block; the large size of these blocks, each containing two floating-point multipliers, plus a number of floating-point adders, is the limiting factor on how many can be used.

As might be expected, the time per observation is inversely proportional to the number of files processed in parallel, and accordingly, the speedup over software is proportional to that value.

6.7 Results

The findings detailed above can be expressed as follows. Putting aside the use of a language model or a duration model, the times for hardware and software, and the bandwidth, can be expressed as:

$$\begin{aligned} T_s &= K_s \cdot (H \cdot M) \\ T_h &= K_h \cdot (H \cdot M)/(H^* \cdot M^* \cdot F^*) \\ B_h &= (W_m + W_v) \cdot (H^* \cdot M^*) \qquad (6.1) \end{aligned}$$

where T_s and T_h are the times for software and hardware respectively, and B_h is the hardware bandwidth; K_s and K_h are constants dependent on the speed and processing power of the processor and hardware device, respectively; W_m and W_v represent the width of the mean and the variance constant; H and M are the numbers of HMMs and mixture components, with H^* and M^* being the number of each implemented in parallel; and F^* is the number of files processed in parallel in hardware.

The speedup of hardware over software is therefore:

$$T_s/T_h = (K_s/K_h) \cdot H^* \cdot M^* \cdot F^* \tag{6.2}$$

which suggests that speedup is independent of the number of HMMs and mixture components.

For the language model, which involves computing the most likely predecessor for each HMM, the results for software (T_{sls}), hardware processing all HMMs in parallel (T_{hlp} and B_{hlp}), and hardware processing one HMM at a time in serial (T_{hls} and B_{hls}) are:

$$\begin{aligned} T_{sls} &= K_{sls} \cdot H^2 \\ T_{hlp} &= K_{hlp} \\ B_{hlp} &= K_{blp} \cdot H \\ T_{hls} &= K_{hls} \cdot H^2 \\ B_{hls} &= K_{bls} \end{aligned} \tag{6.3}$$

Note that the time for the parallel hardware implementation, T_{hlp}, is actually $K_{hlpp} \cdot H$, but as it operates in parallel with the scaler, and takes a similar number of clock cycles, its latency is hidden, resulting in the constant K_{hlp} being small.

Accordingly, the parallel and serial speedups are:

$$\begin{aligned} T_{sls}/T_{hlp} &= (K_{sls}/K_{hlp}) \cdot H^2 \\ T_{sls}/T_{hls} &= (K_{sls}/K_{hls}) \end{aligned} \tag{6.4}$$

showing that the speedup for a parallel implementation is proportional to H^2, while for a serial implementation, the speedup is unconnected to the number of HMMs.

For the duration model, in software (T_{sd}) and hardware (T_{hd} and B_{hd}):

$$\begin{aligned} T_{sd} &= K_{sd} \cdot D \\ T_{hd} &= K_{hd} \cdot (D/D^*) \\ B_{hd} &= K_{bd} \cdot D^* \end{aligned} \tag{6.5}$$

where D is maximum duration, and D^* is the number of duration stages implemented in parallel. Hence the speedup is independent of the maximum duration:

$$T_{sd}/T_{hd} = (K_{sd}/K_{hd}) \cdot D^* \tag{6.6}$$

6.8 Summary

In this chapter, the effect that the model parameters can have on processing time and bandwidth have been examined, and these relationships are summarised in Table 6.1. Table 6.2 summarises the corresponding effects on the speedup of hardware over software.

Following on from this, the next chapter presents the results of the various implementations, and explores whether the predictions made in this chapter are seen in practice.

Table 6.1: Summary of effects on time per observation and bandwidth

	Num. HMMs (SW)	Num. HMMs (HW)	Num. HMMs impl. (HW)
Time/obs	proportional	proportional	inv. prop.
Bandwidth		**independent**	proportional

	Num. mixtures (SW)	Num. mixtures (HW)	Num. mixtures impl. (HW)
Time/obs	proportional	proportional	inv. prop.
Bandwidth		**independent**	proportional

	Language model (HW – parallel)	Language model (HW – serial)	Language model (SW)
Time/obs	no increase	increase $\propto$ (num. of HMMS)^2	increase $\propto$ (num. of HMMs)^2
Bandwidth	$\propto$ num. of HMMS, if not stored on-chip	**independent** of num. of HMMs	

	Max duration (SW)	Max duration (HW – serial)	Num stages impl. (HW)	Num. parallel files (HW)
Time/obs	proportional increase	proportional increase	inv. prop. increase	inv.prop.
Bandwidth		**independent**	proportional	**independent**

Table 6.2: Summary of effects on speedup

	Speedup (hardware vs. software)
Num. HMMs	Independent
Num. mixture components	Independent
Num. HMMs impl.	Proportional
Num. mixture components impl.	Proportional
Num. of duration stages impl.	Proportional
Num. parallel files	Proportional
Max duration – serial	increase independent of max duration
Language model – parallel	increase $\propto$ (num. of HMMs)^2
Language model – serial	increase independent of num. of HMMs

Nourishing a youth sublime

With the fairy tales of science, and the long result of Time.

Alfred, Lord Tennyson (1809–92)

7 Results

Having designed and implemented the FPGA designs and the system software, and having verified that the recognition results resemble those of HTK, it is now necessary to evaluate the hardware's applicability to this problem.

Two criteria govern this applicability: speed (i.e. can it outperform software running on a powerful computer) and cost (initially in terms of resources, but ultimately economically).

7.1 Hardware vs software

These tests compare the rates at which the FPGA and software can perform Viterbi decoding, incorporating observation probability computation for all but the discrete HMM implementation. HTK was not timed, as it was run on a UNIX workstation, so any timing comparison with a PC would not have been particularly enlightening.

In Tables 7.1, 7.2, 7.4 and 7.5, the first set of columns shows, for each implementation, the number of HMMs in the model, the number of elements in each feature vector

(for continuous HMM models), the number of mixture components, the number of speech files processed simultaneously in hardware, and the maximum duration for those implementations which use duration modelling (for the rest, D is effectively 1).

The next two columns indicate the average time taken in microseconds to process one observation. This is computed across the whole corpus of 160 test files, totalling around 50,000 observations. For the software tests, the timer is started once a speech file has been loaded, and stopped when all of its observations have undergone observation probability computation and Viterbi decoding; backtracking and file operations are not included in the total, as they are the same for both hardware and software.

For the hardware, the timer is started after the observation files have been written to the RC1000's RAM, and immediately prior to the host releasing control of the RAM. The FPGA monitors this and begins processing as soon as the RAM is released. Once the FPGA has finished with its files, it releases the RAM, which the PC then notices, and the timer is stopped.

For those implementations which process three speech files in parallel, the total number of observations processed is more than the actual total number of observations in the files. This is because the FPGA cannot release the RAM until it has finished working on the longest of the three files currently loaded. While this is going on, the shorter files are effectively "padded" with dummy data values in order to make them the same length as the longest file. The result is that the average time per observation is slightly lower than it would otherwise be, because the dummy data values are taken into account, but does represent the timing when the system is running at maximum capacity.

Following the values for time per observation are the speedups over real time. Because of the slow movement of the vocal tract, it is sufficient to produce observation feature vectors only once every 10 ms.

Next is the FPGA clock frequency in megahertz. This is the value reported by the

implementation tools as being the maximum speed at which the FPGA can be reliably operated when configured with the current design.

The final set of columns compares hardware with software. The first value is the actual speedup, with the FPGA running at the clock speed shown. The second is the speedup that would be obtained if a rate of 50 MHz were used; this value was chosen to make it easier to compare the three central implementations (Cont1P, Cont3P and Cont3_4), 50 MHz being the speed at which the fastest of them operates (80 MHz is used for the same reason for the projected Virtex-II values). Finally, these values are shown normalised to Cont1P, again to allow them to be more easily compared.

7.1.1 Virtex/Virtex-E vs Pentium-III 450 (debug mode)

The first set of tests compares the hardware implementations (Disc1 and Cont1 on the XCV1000, the others on the XCV2000E) with the software running on the PC which houses the RC1000 boards, a Pentium-III 450.

For each implementation, these are the first tests performed on the system, and so the software is compiled in debug mode. Debug mode is essential during development, as it makes it much easier to locate bugs and track execution. The only downside is that the executable is significantly slower than its release-mode counterpart, but the data is included here nonetheless because it represents the first successful set of tests for each implementation, and also verifies some of the predictions made in chapter 6, which is not the case with some of the later tests (specifically, Cont1/Cont1P in software).

The results of these tests are shown in Table 7.1. Cont3_4D was not implemented in hardware because of its unsuitability for this medium, as mentioned in section 5.8, so the times shown are for software only, with three different values of the maximum duration D, rather than the single value proposed for hardware. Although a software implementation of this model was not a major aim of the research, it did provide a testbed for the method

of computation that would be used in hardware, and also allowed the prediction about the relationship between D and processing time, mentioned in section 6.5, to be verified.

Disc1 and Cont1 illustrate the additional computational burden imposed by having to handle Gaussian distributions. The difference in times for the hardware is more marked than for software, not least because the larger design results in a slower clock speed.

Comparing Cont1 and Cont1P, we can see that the normalised speedup demonstrates, as might be expected, that processing three files in parallel results in a system that is three times faster.

The increase in times for Cont1P, Cont3P and Cont3_4 is roughly in line with that predicted in chapter 6. Taking Cont1P as a baseline, Cont3P would be expected to give software and hardware times of 70,000 and 509 μs respectively, these values computed by scaling the number of HMMs from 49 to 634; this compares to the actual values of 73,500 and 769. Cont3_4 would produce 193,000 and 1,410 μs, found by scaling the number of HMMs in the same way, the number of elements from 39 to 27, and the number of mixture components from 1 to 4; the actual values are 190,000 and 1,650 μs.

Furthermore, the normalised speedup values are roughly the same, as expected, suggesting that speedup is indeed independent of the number of HMMs, elements or mixture components.

Cont3_4D, tested in software, was run with three different values for the maximum duration D, in order to test the prediction that the time *added* by the duration modelling (but not the total time per observation) is proportional to D. The prediction appears to hold: the numbers suggest that an increase in D of 15 leads to an increased time of around 50,000 μs, equivalent to an increase of 3,300 μs for each increment of D.

The key results from this test, though, are the actual times for hardware, and their speedups over real time and software. We see that all of the FPGA implementations are faster than real time, with Cont3_4D, the most complex one, having the lowest value of

Table 7.1: Timing: Virtex/Virtex-E vs Pentium-III 450 (debug mode)

Name	HMMs	El	Mix	Files	D	Time/obs (μs) SW	Time/obs (μs) HW	vs real time SW	vs real time HW	Clk MHz	HW v SW actual	HW v SW 50 MHz	HW v SW norm.
Disc1	49		1	1		886.9	2.029	11.28	4929	55	437.2	397.4	2.903
Cont1	49	39	1	1		5386	134.2	1.857	74.50	44	40.13	45.60	0.3331
Cont1P	49	39	1	3		5386	39.35	1.857	254.1	50	136.9	136.9	1.000
Cont3P	634	39	1	3		73504	768.9	0.1360	13.01	33	95.59	144.8	1.058
Cont3_4	634	27	4	3		190389	1652	0.0525	6.055	43	115.3	134.0	0.9792
Cont3_4D1	634	27	4	3		190389	17529	0.0525	0.5705	62	10.86	8.759	0.0640
Cont3_4D	634	27	4	3	15	240441		0.0416	Not implemented				
					31	292034		0.0342	Not implemented				
					63	397276		0.0252	Not implemented				

Table 7.2: Timing: Virtex/Virtex-E vs Pentium-III 450 (release mode)

Name	HMMs	El	Mix	Files	D	Time/obs (μs) SW	Time/obs (μs) HW	vs real time SW	vs real time HW	Clk MHz	HW v SW actual	HW v SW 50 MHz	HW v SW norm.
Disc1	49		1	1		59.72	2.029	167.4	4929	55	29.44	26.76	4.311
Cont1	49	39	1	1		244.3	134.2	40.94	74.50	44	1.820	2.068	0.3331
Cont1P	49	39	1	3		244.3	39.35	40.94	254.1	50	6.208	6.028	1.000
Cont3P	634	39	1	3		5893	768.9	1.697	13.01	33	7.665	11.61	1.871
Cont3_4	634	27	4	3		13266	1652	0.7538	6.055	43	8.032	9.340	1.505
Cont3_4D1	634	27	4	3		Not testable							
Cont3_4D	634	27	4	3	15	17808		0.5616	Not implemented				
					31	22146		0.4516	Not implemented				
					63	30732		0.3254	Not implemented				

just over 6 times real time. In software, only the monophone implementations can process data faster than real time.

The speedups over software are rather greater, with Cont1P, Cont3P and Cont3_4D all being around 100 times faster. Cont1, processing just one file at a time, achieves a speedup less than a third of its parallel equivalent. The results for Cont3_4D1 show simply that offloading the observation probability calculations onto hardware allow a modest speedup over software of just under 11, equivalent to half real time. The smaller design permits a higher clock speed than for the other implementations.

7.1.2 Virtex/Virtex-E vs Pentium-III 450 (release mode)

While debug mode is essential during testing, the software must be compiled in release mode in order to properly evaluate its performance. The results are shown in Table 7.2.

The tests are the same as for debug mode, with the exception that Cont3_4D1 was found not to work in release mode; timing issues between the PC and the RC1000, which caused numerous problems during system development, are believed to be the cause.

The release-mode software is an order of magnitude faster than its debug-mode counterpart, the relative speedup ranging from 12.5 to 15. The exception to this is Cont1/-Cont1P, which is 22 times faster, possibly to due to the processor's cache coming into play; Cont3P and Cont3_4D require more data to be stored and processed, which may not make caching possible to the same extent.

As a result, while Cont3P and Cont3_4 have similar speedups (though not as close together as for debug mode), they are larger to an unexpected extent than that of Cont1P. Cont1P does however remain almost exactly three times faster than Cont1.

When comparing actual versus expected times for the software, Cont1P's deviation adversely affects any predictions when using it as the baseline: Cont3P would give predicted times of 3,160 μs (compared to 5,890), and Cont3_4, 8,750 μs (compared to 13,300). Us-

Table 7.3: Release dates for FPGAs and processors under test [62][64][71]

Year	FPGA	Processor
1998	Virtex	
1999	Virtex-E	Pentium-III 450
2000		
2001	Virtex-II	
2002	Virtex-II Pro	XP 2000+

ing Cont3P instead gives a time for Cont3_4 of 16,300 μs; Cont1P would be expected to take 455 μs, rather than 244.

Duration modelling again shows a linear increase, with a delay of 4,200 μs for a change in D of 15, or 280 μs per increment of D.

While the hardware processing rates do not change for these tests, the software is operating faster. Hence the hardware/software speedup is reduced to the range 6 to 8 for Cont1P, Cont3P and Cont3_4. Cont3P is now faster than real time in software.

7.1.3 Virtex/Virtex-E vs Athlon XP 2000+ (release mode)

As a result of the fast pace of the electronics world, illustrated in Table 7.3, neither the PC nor the FPGAs can be considered as top-of-the-range any more. In order to better assess what a modern PC could do, the software was re-run, in release mode, on a different PC, containing an AMD Athlon XP 2000+ processor (nominally equivalent to an Intel Pentium 4 running at 2 GHz, though its clock speed is actually 1.67 GHz). These results are shown in Table 7.4. Note that Cont3_4D1 (observation probabilities in hardware, decoding in software) could not be tested in this way, as it requires the RC1000 to be present in the PC under test.

Again we see Cont1P behaving in an unusual manner. Comparing the speedup of the Pentium and Athlon, the latter is 2.5 to 3.5 times faster (not unusual for a processor that

Table 7.4: Timing: Virtex/Virtex-E vs Athlon XP 2000+ (release mode)

Name	HMMs	El	Mix	Files	D	Time/obs (μs)		vs real time		Clk	HW v SW		
						SW	HW	SW	HW	MHz	actual	50 MHz	norm.
Disc1	49		1	1		20.06	2.029	498.5	4929	55	9.888	8.989	7.040
Cont1	49	39	1	1		50.25	134.2	199.0	74.50	44	0.374	0.425	0.3331
Cont1P	49	39	1	3		50.25	39.35	199.0	254.1	50	1.277	1.277	1.000
Cont3P	634	39	1	3		1726	768.9	5.794	13.01	33	2.244	3.401	2.663
Cont3_4	634	27	4	3		5125	1652	1.951	6.055	43	3.103	3.608	2.826
Cont3_4D1	634	27	4	3		Not testable							
Cont3_4D	634	27	4	3	15	6524		1.533	Not implemented				
					31	8032		1.245	Not implemented				
					63	9919		1.008	Not implemented				

Table 7.5: Timing: Virtex-II (projected) vs Athlon XP 2000+ (release mode)

Name	HMMs	El	Mix	Files	Time/obs (μs)		vs real time		Clk	HW v SW		
					SW	HW	SW	HW	MHz	actual	80 MHz	norm.
Cont1P	49	39	1	3	50.25	46.24	199.0	216.3	43	1.087	2.043	1.000
Cont3P	634	39	1	3	1726	454.2	5.794	22.02	56	3.800	5.441	2.663
Cont3_4	634	27	4	3	5125	887.7	1.951	11.26	80	5.774	5.774	2.826

is rated at 4 times the speed of the Pentium 450) — but for Cont1/Cont1P, it is nearly 5 times faster.

Looking at the normalised hardware/software speedups, Cont3P and Cont3_4 have similar values. However, the gap between these and Cont1P is even wider than before. Given that the code in all cases does not change, it again appears that use of the cache is the most likely cause. Irrespective, Cont1P is once more three times faster than Cont1, that latter being unique, in that on this occasion, the software outperforms the hardware!

Performing the same comparisons as before, with Cont1P as the baseline, the predictions are again skewed: Cont3P would take 650 μs rather than the actual 1,726, and Cont_4 would take 1,800 μs rather than 5,125. With Cont3P as baseline, Cont_4's predicted value is a more sensible 4,779 μs; Cont1P would be expected to take 143 μs, not 50.3.

The values for duration modelling diverge slightly from the linear, but approximate figures are 1,150 μs for a change in D of 15, or 76 μs for a single increment.

On this processor, the software is approaching the processing power of the hardware; the speedups are now in the range 1.3 to 3. In addition, all of the models are at least as fast as real time when run in software.

7.1.4 Virtex-II (projected) vs Athlon XP 2000+ (release mode)

Comparing a Virtex-E to an Athlon XP is not a particularly fair test, as the FPGA is three years older than the processor. At the time of writing, the fastest Virtex-class FPGA family is the Virtex-II Pro. The software tools for this being unavailable, Cont1P, Cont3P and Cont3_4 were re-targeted to the slightly slower Virtex-II, with only the minimum of changes to the design. Based on the maximum clock speed for an XC2V4000-5 reported by the implementation tools, and in the absence of an actual chip on which to test the designs, timing values have been estimated for these designs. These are compared to the high-end PC values in Table 7.5.

The results suggest that the FPGA can outperform the Athlon by a factor of 5.8 for the most complex speech model. The actual speedup is lower for the other two models under test. When normalised relative to each other, the values are the same as for the Virtex-E tests, since in the course of the calculations, the clock rates are cancelled out.

The reported maximum clock speeds are lower than expected. 80 MHz for Cont3_4 is respectable, but the Virtex-II can go faster. 56 MHz for Cont3P is approaching double the speed of the Virtex-E implementation, but is low for the newer chip. The fact that Cont1P produces a rate which is slower than that of the Virtex-E is especially unusual.

Although these three designs were originally aimed at the Virtex-E, the only parts that are device-specific are the cores, generated by Xilinx's Core Generator, which include multipliers and memory blocks. As such, these had to be regenerated specifically for the Virtex-II implementations. Shift registers were correctly inferred by the synthesis software for both the Virtex and Virtex-II.

While it is possible that modifications to the VHDL code (and, if necessary, floor-planning), in order to take advantage of the Virtex-II's features, may yield faster designs, it was felt that to do so was not a productive use of the time available, given the unavailability of a real device on which to test the design.

7.2 Resource usage

The resource usages for the various implementations are shown in Table 7.6. The Virtex and Virtex-E families share the same architecture, but just have different array dimensions and quantities of Block RAM, so direct resource comparisons are valid. This is, however, balanced by the changes to the design that took place after each implementation, which tended to result in space being saved in one block, only for it to be filled when another block was added to the design.

The amount of on-chip memory used is indicative of the quantity of data it is necessary to have available when doing speech recognition. This is exemplified by Cont3_4, which uses all of the Virtex-E's Block RAM, in addition to distributed RAM — and that does not even include the mean and variance values for the observation probability computations, nor the predecessor information produced by the decoding process, all of which have to be stored off-chip for lack of space!

Besides the bandwidth issues detailed earlier, this illustrates another difficulty when it comes to implementing algorithms like this in hardware: whatever resources you have can easily be used up just by employing a more complex model.

7.3 Recognition rates

As this research is about implementing recognition algorithms and assessing their applicability to hardware, accuracy is not a priority in these implementations. It is sufficient for the models employed to be merely *representative* of real models — namely, for them to be of similar size and complexity, though not necessarily containing the same numbers.

As previously mentioned, this is of particular relevance for duration modelling. No data values are available for the p.d.f $d_j(\tau)$, so instead they are generated, based on the otherwise implicit duration model, such that $d_j(\tau) = a_{jj}^{\tau-1}$. The effect of this is to reduce the accuracy of the more complex model; but because it is representative in its complexity, that is enough.

For completeness, the accuracy figures for each model are given in Table 7.7. The results for the hardware and software are shown, along with those for HTK, which was taken as the benchmark. All of the accuracy statistics were generated using HTK's results tool.

The columns in the table are as follows. s and p are the grammar scale factor and word

Table 7.6: Resource usage

Name	HMMs	El	Mix	Files	D	FPGA	Clk	Slices		FFs		LUTs			Block RAM	
							MHz	Total	%	Total	%	Total	%	SRLs	Total	%
Disc1	49		1	1		XCV1000	55	1583	12	2113	8	1900	7	216	8	25
Cont1	49	39	1	1		XCV1000	44	5599	45	6451	26	7827	31	615	17	53
Cont1P	49	39	1	3		XCV2000E	50	13990	72	14807	38	19176	49	2103	30	18
Cont3P	634	39	1	3		XCV2000E	33	14793	77	18097	47	21712	56	1395	138	86
Cont3_4	634	27	4	3		XCV2000E	43	13618	70	17201	44	18826	49	965	160	100
Cont3_4D1	634	27	4	3		XCV2000E	62	10137	52	14256	37	14307	37	965	25	15
Cont3_4D	634	27	4	3	15	Not implemented										

Table 7.7: Recognition rates. The results shown for each model are those whose values of s and p produce the highest accuracy figure. s = grammar scale factor, p = word insertion penalty, H = number of correct labels, D = number of deletions, S = number of substitutions, I = number of insertions, N = total number of labels. %Correctness = H/N×100%, Accuracy = (H−I)/N×100%

Name	HMMs	El	Mix	Files	D	s	p	Platform	H	D	S	I	N	%Corr	Acc
Disc1	49		1	1		1	0	HTK	2121	895	2575	840	5591	37.94	22.91
								S/W	2120	896	2575	838	5591	37.92	22.93
								H/W	2120	896	2575	838	5591	37.92	22.93
Cont1	49	39	1	1		6	0	HTK	2913	959	1719	145	5591	52.10	49.51
								S/W	2912	961	1718	145	5591	52.08	49.49
								H/W	2912	961	1718	145	5591	52.08	49.49
Cont1P	49	39	1	3		6	0	HTK	2913	959	1719	145	5591	52.10	49.51
								S/W	2912	961	1718	145	5591	52.08	49.49
								H/W	2912	961	1718	145	5591	52.08	49.49
Cont3P	634	39	1	3		6	0	HTK	2970	1025	1540	173	5535	53.66	50.53
								S/W	2969	1024	1542	171	5535	53.64	50.55
								H/W	2969	1024	1542	171	5535	53.64	50.55
Cont3_4	634	27	4	3		5	0	HTK	2922	1144	1469	140	5535	52.79	50.26
								S/W	2909	1174	1452	131	5535	52.56	50.19
								H/W	2909	1174	1452	131	5535	52.56	50.19
Cont3_4D1	634	27	4	3		5	0	HTK	2922	1144	1469	140	5535	52.79	50.26
								S/W	2909	1174	1452	131	5535	52.56	50.19
								H/W	2909	1174	1452	131	5535	52.56	50.19
Cont3_4D	634	27	4	3	15	5	0	S/W	3028	539	1968	635	5535	54.71	43.23
					31				3000	572	1963	599	5535	54.20	43.38
					63				3001	573	1961	588	5535	54.22	43.60

insertion penalty respectively, as described in section 2.4. These were varied in order to find the values which produced the highest accuracy figures.

The results given are computed across all 160 test files. H is the number of labels (monophones or biphones/triphones, as appropriate) correctly identified. D is the number of deletions, i.e. the number of labels missed out. S is the number of substitutions, as when the recogniser has correctly realised that something has been uttered, but has misidentified it. I is the number of labels erroneously inserted. N is the total number of labels. This last value is slightly less for the biphone/triphone models, as the length of one of the test files was such that there was insufficient memory on the RC1000 to store all of its predecessor information, so it was omitted.

Percentage correctness is computed as $\frac{\mathrm{H}}{\mathrm{N}} \times 100\%$, and so does not take into account any insertions. Accuracy does, and is equal to $\frac{\mathrm{H-I}}{\mathrm{N}} \times 100\%$.

As expected, the discrete monophone model (Disc1) has a very low accuracy rate, which is improved significantly by switching to continuous HMMs (Cont1/Cont1P). While we would expect the use of biphones and triphones (Cont3P) to increase accuracy further, there is only a slight increase. This is most likely due to a lack of training data, as the more complex the model, the more values that are required to train it successfully.

The same goes for the model incorporating mixture components (Cont3_4/Cont3-_4D1), which has a marginal drop in accuracy. The artificially generated data values used for duration modelling (Cont3_4D) lead to a further drop. The results of this model are for three different values of D, processed solely in software, as HTK cannot handle duration modelling, and there is no hardware implementation. Increasing D leads to a slight increase in accuracy, as longer instances of a particular utterance (commonly silence) can be correctly handled.

In all cases, the results from the hardware and software match perfectly. This is because they are designed to perform the calculations in the same way, with the software

limited to the same bit widths as the hardware.

There are a small number of discrepancies compared to HTK, most likely due to differences in computation methods, bit widths, and possibly rounding methods as well.

It should be noted that although the accuracy scores are low compared to commercial recognisers or those described in chapter 3, this is not a product of the implementations. Rather, each of the models used is limited by its ability to successfully model speech, and additionally by any shortage of suitable training data. So the former establishes a range of likely values for the accuracy rate, and the latter dictates where in that range the actual number is. Whatever the result, it is due to these two facets of the models, and not the implementations which use them.

7.4 Summary

In this chapter, the results produced by the implementations have been presented. Timing information has been provided for the FPGAs used, along with projected values for another one. The corresponding software times have also been given for two PCs. Some of the predictions made in the requirements analysis have been confirmed. Resource usage has been described, and recognition rates have been outlined, including those produced by HTK in order to verify the data produced by the implementations.

The final chapter follows, drawing on the results given here and in previous chapters, to form a set of key conclusions.

To define it rudely but not inaptly, engineering... is the art of doing that well with one dollar, which any bungler can do with two after a fashion.

Arthur Mellen Wellington (1847-1895)

8 Conclusions

In this thesis, the implementation of speech recognition in programmable logic has been investigated. Having described relevant parts of speech recognition theory, and given examples of previous hardware recognisers, original designs for the decoder part of the system have been presented. These have included the ability to process multiple speakers or speech files in parallel despite bandwidth restrictions, and a novel implementation of the log-add algorithm for Gaussian mixture summation.

In addition, the accompanying multi-purpose software toolkit has been described, without which an operational hardware implementation would have been impossible, as well as a VHDL interface for the FPGA development board, which has since been used in other people's designs.

Details have been given of how these designs were implemented on an FPGA, and the results of these implementations analysed, which included comparing them with software equivalents. An analysis of the requirements of such a system has also been presented, whether that system be in hardware or software, in terms of the parameters that govern the complexity of the model, and their effect on speed and bandwidth.

The principal aims and objectives outlined in section 1.1 were to design and implement a speech recognition system, with the decoder stage implemented in hardware, in order to ascertain the suitability of doing so, the speedup of hardware over software, and the requirements of a hardware speech recogniser.

With reference to these objectives, the key conclusions are summarised below.

8.1 Key conclusions

8.1.1 Suitability

Gaussian mixture computation (which is the most computationally expensive part of the recogniser) and the Viterbi decoder both have the potential for significant parallelism, but this is limited in practice by available bandwidth and logic resources.

However, processing multiple speakers or speech files in parallel provides an alternative method for exploiting the capabilities of hardware. The log-add algorithm can also be implemented in a particularly efficient fashion in hardware.

Duration modelling is better suited to a SIMD array with local memory than an FPGA. A full language model cannot feasibly be implemented on an FPGA at present, and may also suit a SIMD array.

8.1.2 Hardware vs software

Simply put, hardware is faster than software. The speedups found when comparing devices of similar age suggest modest speedups of less than 10. But if an expensive FPGA can do the job of half a dozen PCs, then savings can be made.

The speed boost given to the continuous monophone implementations (Cont1/Cont-1P), but not to the biphone/triphone ones, probably by the processor's cache, was unexpected. If the cache is responsible, the benefits it provided were dependent on the amount

of data being processed being below a certain size. While the processors used cannot take advantage of the larger models in this way, it is possible that future ones could.

An FPGA (or ASIC) can benefit from advances in technology in ways that a processor cannot. While processors can run faster, and nowadays may have a constrained degree of parallelism, a dedicated device's dependency on resources and bandwidth means that if these two factors are increased, the number of HMMs, etc, implemented in parallel can also be increased, leading to processing times much faster than can be achieved by a higher clock speed alone.

Whatever the speedups of an FPGA over a PC, it should be remembered that if an FPGA design were to be converted into an ASIC, we could expect the ASIC to be another order of magnitude faster.

8.1.3 Requirements

The requirements analysis of chapter 6 was able to successfully predict ball-park figures for hardware and software observation times, factoring in the number of HMMs, elements, mixture components, and files processed in parallel. Some predictions were, however, skewed by the software processing its data faster than expected in certain circumstances, possibly due to the aforementioned cache effects.

As predicted, the speedup of hardware over software was dependent on the number of files processed in parallel, and was independent of the size of the model.

Duration modelling in software also behaved as expected, giving a linear increase in processing time as the maximum duration was increased.

8.2 Summary

What has been demonstrated is that hardware is a viable platform for a speech recognition system. Some parts are well-suited for an FPGA, while others fare better with different architecture types, or are best processed in software.

It has also been shown that an FPGA can outperform software, and that the speedup is more dependent on bandwidth and resources than on the size of the recognition model.

8.3 Further work

This speech recognition system has plenty of scope for expansion. Just using the algorithms outlined here, the number of HMMs could be increased, as could the number of mixture components. If duration modelling were to be implemented in hardware, perhaps using a SIMD array rather than an FPGA, that could then allow the maximum duration to be raised, and the duration probability distribution made more complex.

HMMs can also incorporate additional types of data at the decoding stage, further improving the model, while not compromising the HMM principle of delayed decision making. The language model is one such extension. It carries with it a potentially large increase in either processing time or resources, but also increased accuracy. Such a block would simply take the place of the existing scaled-down language model block that already exists.

There is the possibility of this part of the system being run in software on a faster processor or microcontroller instead. While this is technically feasible, there would be a number of issues to resolve. Firstly, bandwidth: the hardware outputs $\delta_{t-1}(I)$ at the rate of one per clock cycle, along with one value from each of the H model data stores. It subsequently requires two data values per clock cycle, ideally in the time taken for the scaler to do its processing, the number of clock cycles equalling the number of HMMs. Secondly,

the data transfers would need to be synchronised somehow, most simply done by using a shared memory, as is the case for the FPGA–PC data transfers in the implementations described here.

Segmental modelling [13][41] is another method of improving accuracy, whereby states are associated with sequences of feature vectors, with the relationship between successive feature vectors being modelled by a trajectory through the feature space (duration modelling is a special case of this). Once again, increased accuracy is balanced by significant additional computation.

It has already been mentioned that an FPGA is being used here as a prototyping platform, with the idea being that if a commercial version were to exist, it would use an ASIC instead. Because reconfiguration is not used, the only significant change to the design (besides converting any FPGA-specific primitives to their ASIC equivalent) would be to replace the on-chip look-up tables containing the transition and between-HMM probabilities with RAM, with the idea that these values would be loaded onto the chip at reset. This would allow the system to handle different speech models and different languages.

Whereas software can simply allocate as much memory as needed when a new model is used, hardware has to have its parameters fixed at compile time. Hence if support for multiple models were required, the maximum allowable number of HMMs would have to be decided in advance. Any model which had fewer may then have to have dummy HMMs/utterances added, whose corresponding probabilities would be set so as to make any occurrence of the utterance impossible.

One final consequence of using an ASIC is the growing speed gap between ASICs and RAM. Though the implementations described here operated at a clock rate which the RAM could handle, the Virtex-E is capable of outperforming the RC1000's on-board memory, and so the problem is likely to be compounded with faster FPGAs, and more so with ASICs. The bandwidth could however be maximised by using a chip package with

more pins, and arranging for a slower chip and faster memory, as long as this would meet any processing rate criteria.

There is also the on-chip RAM to consider. On an FPGA, the interconnect delays dominate the timing. These are considerably reduced on an ASIC, but on-chip RAM (SRAM for an FPGA) is likely to behave much the same on both types of chip, leading to another potential bottleneck unless the memory can keep up.

While software or DSPs have been sufficient to deal with the speech pre-processing, and although software is well-suited to the post-decoding dictionary look-up, future research may also want to look again at the pre- and post-processing stage, towards the goal of a complete integrated speech recognition chip.

8.4 The future...

This thesis begins with the idea that, in time, we will be able to talk to a computer in the same way we might talk to a human being, and expect it to respond similarly.

The artificial intelligence side of that vision still seems a long way off. As for the speech recognition part: desktop PCs can already do dictation in real time, if only under "ideal" conditions. As PCs become ever more powerful, recognition software can be improved. Whether or not we will see a speech co-processor in a PC will depend on whether or not a future PC will be powerful enough to handle the additional algorithmic complexity, which is necessary to make recognition software as robust and flexible as users may require.

However, for environments where we would like one chip to do the work of several PCs, or for mobile applications, speech recognition in hardware could prove to be very useful. While PCs will continue to provide greater and greater processing power, devices such as FPGAs will benefit both from an increase in speed, and an increase in

on-chip resources — the latter providing designers with the additional processing fabric and bandwidth necessary for more complex algorithms and hence greater recognition accuracy, as well as greater parallelism. Though FPGAs are currently too power-hungry for use in mobile devices, that could yet change; until then, ASICs will retain their monopoly in that domain.

Given the incredible rate of progress in the electronics sector, it is always dangerous to make predictions about the future. Nonetheless, as far as FPGAs are concerned, the trend seems to be towards integrating the reconfigurable fabric of FPGAs, and the dedicated logic of ASICs, on to a single device — the system on a chip.

As to how this may affect speech recognition and related applications, and the nature and direction of technology to come: we will just have to wait and see...

If I have seen further... it is by standing upon the shoulders of giants.

Sir Isaac Newton (1643–1727)

References

[1] Alexandres, S., Moran, J., Carazo, J. & Santos, A., "Parallel architecture for real-time speech recognition in Spanish," *Proc. IEEE International Conference On Acoustics, Speech And Signal Processing (ICASSP '90)*, 1990, pp.977–980. 33

[2] Andraka, R., "A survey of CORDIC algorithms for FPGA based computers," *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '98)*, 1998, pp.191–200 *and* `http://www.fpga-guru.com/papers.htm`. 43

[3] Bisiani, R., Anantharaman, T. & Butcher, L., "BEAM: an accelerator for speech," *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '89)*, 1989, pp.782–784. 32

[4] Bohez, E.L.J. & Senevirathne, T.R., "Speech recognition using fractals," *Pattern Recognition*, **34**, No.11, 2001, pp.2227–2243. 42

[5] Bridle, J.S., Brown, M.D. & Chamberlain, R.M., "An algorithm for connected word recognition," *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '82)*, 1982, pp.899–902. 20

[6] Bridle, J.S., Brown, M.D. & Chamberlain, R.M, "Continuous connected word recognition using whole word templates," *Radio and Electronic Engineer*, **53**, No.4, 1983, pp.167–175. 20

[7] Chen, R. & Jamieson, L.H., "Experiments on the implementation of recurrent neural networks for speech phone recognition," *Proc. 30th Annual Asilomar Conference on Signals, Systems and Computers*, 1996, pp.779–782. 42

[8] Cox, S.J., "Hidden Markov models for automatic speech recognition: theory and application," *British Telecom Technology Journal*, **6**, No.2, 1988, pp.105–115. 14, 16, 41

[9] Eldredge, J.G. & Hutchings, B.L., "RRANN: a hardware implementation of the back-propagation algorithm using reconfigurable FPGAs," *Proc. IEEE International Conference on Neural Networks / IEEE World Conference on Computational Intelligence*, **4**, 1994, pp.2097–2102. 42

[10] Frostad, K., "The state of embedded speech," *Speech Technology Magazine*, Mar/Apr 2003 *and* `http://www.speechtechmag.com/`. 41

[11] Gómez-Cipriano, J.L., Pizzatto Nunes, R., Bampi, S. & Barone, D. "Design of functional blocks for a speech recognition portable system," *Proc. 14th Symposium on Integrated Circuits and Systems Design (SBCCI '01)*, 2001, pp.20–25. 29

[12] Gorin, A.L., Riccardi, G. & Wright, J.H., "How may I help you?" *Speech Communication*, **23**, No.1–2, 1997, pp.113–127. 9

[13] Holmes, W.J. & Russell, M.J., "Probabilistic-trajectory segmental HMMs," *Computer Speech and Language*, 1999, **13**, pp.3–37. 115

[14] Holmes, J. N. & Holmes W.J., "Speech synthesis and recognition," Taylor & Francis, 2001. 26

[15] IEE, "FPGAs not ready to go embedded," *IEE Review*, Institution of Electrical Engineers, April 2003. 8

[16] IEEE, "754–1985 IEEE standard for binary floating-point arithmetic", Institute of Electrical and Electronics Engineers, 1985 *and* `http://standards.ieee.org/` 80

[17] James-Roxby, P. & Blodget, B., "Adapting constant multipliers in a neural network implementation," *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM 2000)*, 2000, pp.335–336. 8

[18] Jelinek, F., "Fast sequential decoding algorithm using a stack," *IBM Journal of Research and Development*, 1969, **13**, pp.675–685. 3

[19] Jou, J.M., Shiau, Y.H. & Huang, C.J., "An efficient VLSI architecture for HMM-based speech recognition," *Proc. IEEE International Conference on Electronics, Circuits and Systems (ICECS '01)*, 2001, pp.469–472. 39

[20] Kimball, O., Cosell, L., Schwarz, R. & Krasner, M., "Efficient implementation of continuous speech recognition on a large scale parallel processor," *Proc. IEEE International Conference On Acoustics, Speech And Signal Processing (ICASSP '87)*, 1987, pp.852–855. 35

[21] Krikelis, A., "Continuous speech recognition using an associative string processor," *Proc. IEEE International Symposium on Circuits and Systems*, 1989, pp.183–186. 36

[22] Laufer, R., Taylor, R.R. & Schmit, H., "PCI-PipeRench and the SWORDAPI: a system for stream-based reconfigurable computing," *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '99)*, 1999, pp.200–208. 8

[23] Lee, L.S., Tseng, C.Y., Lin, Y.H., Lee, Y., Tu, S.L., Gu, H.Y., Liu, F.H., Chang, C.H., Hsieh, S.H., Chen, C.H. & Huang, K.R., "A fully parallel Mandarin speech recognition system with very large vocabulary and almost unlimited texts," *Proc. IEEE International Symposium on Circuits and Systems*, 1991, pp.578–581. 4, 33

[24] Lee, S.W. & Hsu, W.H., "Parallel algorithms for hidden Markov models on the orthogonal multiprocessor," *Pattern Recognition*, **25**, No.2, 1992, pp.219–232. 34, 45

[25] Levinson, S.E., "Continuously variable duration hidden Markov models for automatic speech recognition," *Computer Speech and Language*, 1986, **1**, pp.29–45. 27

[26] Lin, M.B., "New path history management circuits for Viterbi decoders," *IEEE Transactions on Communications*, **48**, No.10, 2000, pp.1605–1608. 45

[27] Lin, S. & Costello, D. J., "Error control coding: fundamentals and applications," Prentice-Hall, 1983. 45

[28] Makimoto, T., "The rising wave of field programmability," *Proc. 10th International Conference on Field Programmable Logic and Applications (FPL 2000), Lecture Notes in Computer Science #1896*, 2000, pp.1–6. 7

[29] Makimoto, T., "The hot decade of field programmable technologies," *IEEE International Conference on Field Programmable Technology (FPT 2002)*, 2002, pp.3–6. 7

[30] Melnikoff, S.J., James-Roxby, P.B., Quigley, S.F. & Russell, M.J., "Reconfigurable computing for speech recognition: preliminary findings," *Proc. 10th International Conference on Field Programmable Logic and Applications (FPL 2000), Lecture Notes in Computer Science #1896*, 2000, pp.495–504. 11

[31] Melnikoff, S.J., Quigley, S.F. & Russell, M.J., "Implementing a hidden Markov model speech recognition system in programmable logic," *Proc. 11th International Conference on Field Programmable Logic and Applications (FPL 2001), Lecture Notes in Computer Science #2147*, 2001, pp.81–90. 11, 77

[32] Melnikoff, S.J., Quigley, S.F. & Russell, M.J., "Implementing a simple continuous speech recognition system on an FPGA," *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM 2002)*, 2002, pp.275–276. 11, 80

[33] Melnikoff, S.J., Quigley, S.F. & Russell, M.J., "Speech recognition on an FPGA using discrete and continuous hidden Markov models," *Proc. 12th International Conference on Field Programmable Logic and Applications (FPL 2002), Lecture Notes in Computer Science #2438*, 2002, pp.202–211. 11, 77, 80

[34] Melnikoff, S.J., Quigley, S.F. & Russell, M.J., "Performing speech recognition on multiple parallel files using continuous hidden Markov models on an FPGA," *IEEE International Conference on Field Programmable Technology (FPT 2002)*, 2002, pp.399–402. 11, 82, 83

[35] Melnikoff, S.J. & Quigley, S.F., "Implementing log-add algorithm in hardware," *IEE Electronics Letters*, **39**, No.12, 2003, pp.939–941. 11, 58

[36] Melnikoff, S.J. & Quigley, S.F., "A qualitative analysis of the requirements of performing speech recognition in hardware," submitted to *IEE Proceedings — Computers and Digital Techniques*. 11, 88

[37] Mitchell, C.D., Harper, M.P., Jamieson, L.H. & Helzerman, R.A., "A parallel implementation of a hidden Markov model with duration modeling for speech recognition," *Digital Signal Processing*, **5**, No.1, 1995, pp.43–57 *and* http://purcell.ecn.purdue.edu/~speechg/. 27, 32, 34, 45, 63

[38] Mozer, T., "The third wave: speech in consumer electronics," *Speech Technology Magazine*, Jul/Aug 2000 *and* http://www.speechtechmag.com/. 41

[39] Murveit, H., Mankoski, J., Rabaey, J., Brodersen, R., Stölzle, A., Chen, D., Narayanaswamy, S., Yu, R., Schrupp, P., Schwartz, R., & Santos, A., "Large-vocabulary real-time continuous-speech recognition system," *Proc. IEEE International Conference On Acoustics, Speech And Signal Processing (ICASSP '89)*, 1989, pp.789–792. 31

[40] Nakamura, K., Zhu, Q., Maruoka, S., Horiyama, T., Kimura, S. & Watanabe, K., "Speech recognition chip for monosyllables," *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC 2001)*, 2001, pp.396–399. 37

[41] Ostendorf, M., Digalakis, V.V. & Kimball, O.A., "From HMMs to segment models: a unified view of stochastic modeling for speech recognition," *IEEE Transactions on Speech and Audio Processing*, **4**, No.5, 1996, pp.360–378. 115

[42] Paul, D.B., "An efficient A* stack decoder algorithm for continuous speech recognition with a stochastic language model," *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '92)*, 1992, pp.23–26. 3

[43] Pepper, D.J., Barnwell, T.P. & Clements, M.A., "Using a ring parallel processor for hidden Markov model training," *IEEE Transactions on Acoustics, Speech And Signal Processing*, **38**, No.2, 1990, pp.366–369. 45

[44] Rabiner, L.R., "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, **77**, No.2, 1989, pp.257–286 *and* Waible, A. & Lee, K.F. (eds.), *Readings in Speech Recognition*, 1990, Morgan Kaufmann Publishers, Inc., pp.267–296. 14, 16, 27

[45] Roe, D.B., Gorin, A.L. & Ramesh, P. "Incorporating syntax into the level-building algorithm on a tree-structured parallel computer," *Proc. IEEE International Conference On Acoustics, Speech And Signal Processing (ICASSP '89)*, 1989, pp.778–781. 46

[46] Schmit, H. & Thomas, D., "Hidden Markov modeling and fuzzy controllers in FPGAs," *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '95)*, 1995, pp.214-221. 38

[47] Sezer, S., Heron, J., Woods, R., Turner, R. & Marshall, A., "Fast partial reconfiguration for FCCMs," *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, 1998, pp.318–319. 8

[48] Shi, Y.Y., Liu, J. & Liu, R.S., "Single-chip speech recognition system based on 8051 microcontroller core," *IEEE Transactions on Consumer Electronics*, **47**, No.1, 2001, pp.149–153. 37, 42, 45

[49] Shozakai, M., "Speech interface VLSI for car applications", *Proc. IEEE International Conference On Acoustics, Speech And Signal Processing (ICASSP '99)*, 1999, pp.141–144. 37

[50] Stölzle, A., Narayanaswamy, S., Murveit, H., Rabaey, J.M. & Brodersen, R.W., "Integrated circuits for a real-time large-vocabulary continuous speech recognition system," *IEEE Journal of Solid State Circuits*, **26**, No.1, 1991, pp.2–11. 31, 65

[51] Stogiannos, P., "FCCM coprocessor for real-time continuous speech recognition," Masters Thesis, Microprocessor and Hardware Laboratory, Department of Electronic and Computer Engineering, Technical University of Crete, 1999. 39

[52] Stogiannos, P., Dollas, A. & Digalakis, V., "A configurable logic based architecture for real-time continuous speech recognition using hidden Markov models," *Journal of VLSI Signal Processing Systems for Signal Image and Video Technology*, **24**, No.2–3, 2000, pp.223–240. 11, 39, 43

[53] Sutherland, A.M., Campbell, M., Ariki, Y. & Jack, M.A., "OSPREY: a transputer based continuous speech recognition system," *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '90)*, 1990, pp.949–952. 33

[54] Vargas, F.L., Fagundes, R.D.R. & Junior, D.B., "A FPGA-based Viterbi algorithm implementation for speech recognition systems," *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '01)*, 2001, pp.1217–1220. 39

[55] Viterbi, A.J., "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, **IT-13**, No.2, 1967, pp.260–269. 18, 43

[56] Walther, J., "A unified algorithm for elementary functions," *Proc. Spring Joint Computer Conference*, 1971, pp.379–385 *and* Swartzlander, E.E., "Computer Arithmetic," **1**, IEEE Computer Society Press Tutorial, 1990. 43

[57] Woodland, P.C., Odell, J.J., Valtchev, V. & Young, S.J. "Large vocabulary continuous speech recognition using HTK," *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '94)*, 1994, pp.125–128. 3

[58] Yeh, D., Feygin, G., & Chow, P., “RACER: a reconfigurable constraint-length 14 Viterbi decoder,” *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM ’96)*, 1996, pp.60-69. 44

[59] Young, S., “A review of large-vocabulary continuous-speech recognition,” *IEEE Signal Processing Magazine*, **13**, No.5, 1996, pp.45–57. 14, 15, 16

[60] Young, S., Evermann, G., Kershaw, D., Moore, G., Odell, J., Ollason, D., Povey, D., Valtchev, V. & Woodland, P., “The HTK Book,” Cambridge University Engineering Department, 2002 *and* http://htk.eng.cam.ac.uk/docs/docs.shtml. 4, 5

[61] Yun, H-K., Smith, A. & Silverman, H., “Speech recognition HMM training on reconfigurable parallel processor,” *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM ’97)*, 1997, pp.242–243. 45

[62] Advanced Micro Devices, Inc.
http://www.amd.com/ 102

[63] Celoxica
http://www.celoxica.com/ 70

[64] Intel Corporation
http://www.intel.com/ 102

[65] Martin Russell’s Home Page
(Electronic, Electrical & Computer Engineering, University of Birmingham)
http://www.eee.bham.ac.uk/russellm/ 4, 14

[66] Philips Speech Processing
http://www.speech.philips.com/ 40

[67] Sensory, Inc.
http://www.sensoryinc.com/ 40

[68] Texas Instruments DSP Developers’ Village
http://dspvillage.ti.com/ 41

[69] Telecom Italia Lab (TILAB): System on Chip
http://www.idosoc.com/ 41

[70] TIMIT Acoustic-Phonetic Continuous Speech Corpus
http://www.ldc.upenn.edu/Catalog/LDC93S1.html 75

[71] Xilinx, Inc.
http://www.xilinx.com/ 41, 69, 77, 102

Credits

Supervisor	Dr Steven Quigley
Original research proposal	Dr Philip James-Roxby
Speech model (Cont3_4)	Nicholas Wilkinson
PhD funding	Engineering & Physical Sciences Research Council
Conference funding	University of Birmingham
	Royal Academy of Engineering
	Chinese University of Hong Kong
C++ programming	Microsoft Visual C++ v6.0
VHDL simulation	Aldec Active-HDL v4.2
VHDL synthesis	Synplicity Synplify Pro v7.1
FPGA implementation	Xilinx ISE 4.1
Speech processing	HTK (Hidden Markov Model Toolkit); University of Cambridge
Typsetting language	LaTeX 2_ε
Typesetting engine	MikTeX v2.2
Typesetting environment	TeXnicCenter v1 Beta 6.01
Diagrams	Visio Technical v5.0c
FPGAs	Xilinx Virtex XCV1000 BG560-6
	Xilinx Virtex XCV2000E BG560-6
FPGA development board	Celoxica RC1000-PP (Board v4.0, Logic v1.10)

Speech Recognition in Programmable Logic

A thesis submitted to

The University of Birmingham

for the degree of

Doctor of Philosophy

Written and researched

by

Stephen Melnikoff

Word count: 37,800 (approx.)

He describes a PhD as being, initially at least, “as useful as a chocolate teapot.” As someone who has carried out research in universities before moving to industry, I know a chocolate teapot is an extremely useful product if your customer wants chocolate-flavoured tea.
– Jonathan Fanning, Letters, *IEE News* (Sept. 2002)

©2003 Stephen Jonathan Melnikoff

Appendix: Publications

Letter

Melnikoff, S.J. & Quigley, S.F., “Implementing log-add algorithm in hardware,” *IEE Electronics Letters*, **39**, No.12, 2003, pp.939–941.

International conference papers

Melnikoff, S.J., James-Roxby, P.B., Quigley, S.F. & Russell, M.J., “Reconfigurable computing for speech recognition: preliminary findings,” *Proc. 10th International Conference on Field Programmable Logic and Applications (FPL 2000)*, *Lecture Notes in Computer Science #1896*, 2000, pp.495–504.

Melnikoff, S.J., Quigley, S.F. & Russell, M.J., “Implementing a hidden Markov model speech recognition system in programmable logic,” *Proc. 11th International Conference on Field Programmable Logic and Applications (FPL 2001)*, *Lecture Notes in Computer Science #2147*, 2001, pp.81–90.

Melnikoff, S.J., Quigley, S.F. & Russell, M.J., “Speech recognition on an FPGA using discrete and continuous hidden Markov models,” *Proc. 12th International Conference on Field Programmable Logic and Applications (FPL 2002)*, *Lecture Notes in Computer Science #2438*, 2002, pp.202–211.

Melnikoff, S.J., Quigley, S.F. & Russell, M.J., “Performing speech recognition on multiple parallel files using continuous hidden Markov models on an FPGA,” *IEEE International Conference on Field Programmable Technology (FPT 2002)*, 2002, pp.399–402.

International conference poster

Melnikoff, S.J., Quigley, S.F. & Russell, M.J., “Implementing a simple continuous speech recognition system on an FPGA,” *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM 2002)*, 2002, pp.275–276.